

SCS 2207 - Programming Language Concepts

Data Type



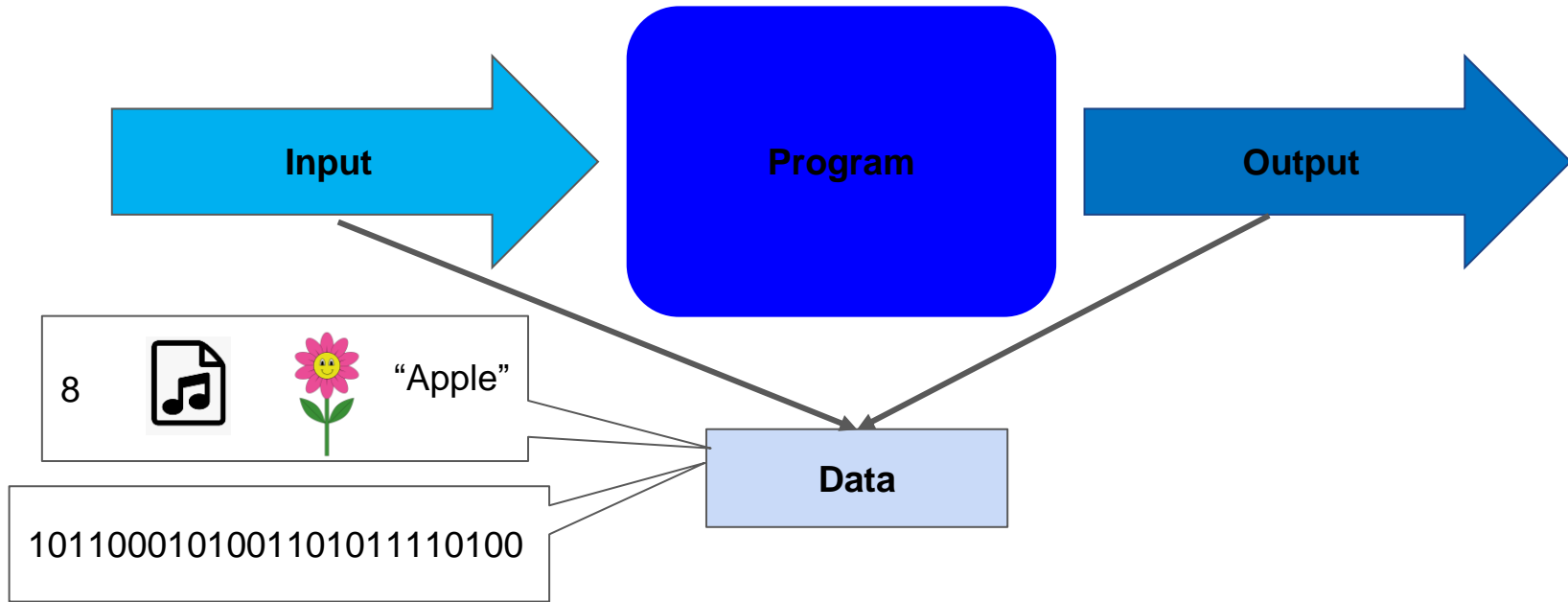
University of Colombo School of Computing

Outline

1. Why Data Types?
2. Common Data Types Provided by the Programming Languages
3. Type Checking

Why Data Types?

Why Data Types?



What is Data Type

- A data type is a **set of values** and a **set of operations** defined on them.
- In data types
 - Data is sorted into **sets**
 - The operations on data are modelled by **functions** on those sets,
 - The tests are modelled by either relations on the sets, or functions on the sets returning a Boolean value.
- A **mathematical model** of a data type called an **algebra of data types**
 - Algebra = Sets + Functions + Boolean-valued Functions

What is Data Types (Cont...)

- Differences among languages exist in
 - The types of data allowed
 - The types of operations available.
 - The mechanisms provided for storing data and for controlling the sequence in which the operations are applied on the data.

Information of **operators** required during language translation

- Only the **signatures of operators** (not their definitions).
 - **Signatures of Operators**
 - Consider the following C sub-program header - `float sub(int x, float y)`
 - What is the signature of this sub-program? - `sub: int × float → float`
- Signature of operations built into the language are automatically available for the translators during translation

Information of **operators** required during language translation

- Every language provides a set of primitive data types built into the language.
- In addition, the language may also provide facilities to allow the programmer to define new data types and operators (via user defined functions).

Types of data

based on who provide the data

- Provided by the hardware.
 - eg: Integer
- Provided by the language.
 - In-built to the language, hence provided by the virtual machine
 - Provided as standard libraries.
 - eg: square root function.
 - Routines/libraries provided by other developers.
- Constructed by the programmer
 - as functions and library routines.
 - as in-line code sequence within the body of the program

Types of data (Cont...)

based on how they are manipulated in a program

- Elementary data objects (Primitive data Types)
 - Data types that are not defined in terms of other types.
 - All programming languages provide a set of primitive data types
 - e.g. integer, boolean, character, real.
 - The set of values defined is usually an **ordered** set with a **least value** and a **greatest value**.
 - Two different values can be compared for larger (smaller) value.
 - Data object always manipulated as a unit.
 - Values can be stored in fixed memory space

Types of data (Cont...)

based on how they are manipulated in a program

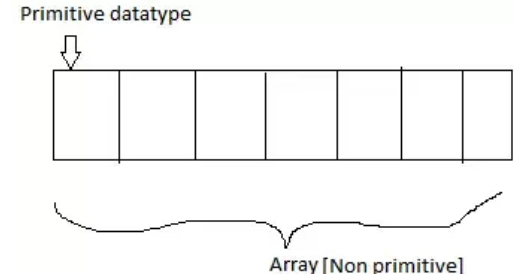
- Elementary data objects (Primitive data Types) - Examples

C/C++	Java	Python (Object Types)
Integer <ul style="list-style-type: none">● short, int, long Real Numbers <ul style="list-style-type: none">● float, double, long Character <ul style="list-style-type: none">● char Pointer	<ul style="list-style-type: none">● boolean● byte Integer <ul style="list-style-type: none">● short, int, long Real Numbers <ul style="list-style-type: none">● float, double Character <ul style="list-style-type: none">● char	<ul style="list-style-type: none">● bool● int● float● complex

Types of data (Cont...)

based on how they are manipulated in a program

- Data structures (Structured or non-scalar)
 - Built out of other types.
 - eg : arrays, records
 - The building blocks of a structured type are its **components**.
 - The types of the components make a structured type.
 - The values of the components make the value of a structured type
 - Aggregation of data object.
 - Require variable memory space to store the values.



Specification of Data Types (Declaration)

Specification of data type (declaration) of a data object provides the following information to the language translation system.

- **Attributes of the data object:** Such as type and name which are usually invariant during its lifetime.
- **Values:** The set of possible values that the data object may contain.
- **Operations:** How the data object may be manipulated.

Specification of Data Types (Declaration)

Declarations may also allow preferred locations of data objects to be specified

Eg:

- In C register variables.
 - `register int c1;`
- In COBOL computational variables,
- USAGE clause - specify the format of a data item/ variable in memory or in file record

Specification of Data Types (Declaration)

Specification of data type facilitate the following

- Static(Dynamic) type checking.
- Selection of correct operations if the operation is overloaded.
- Storage management such as size of the data block, allocation and lifetime.
- Storage representation such as bit representation, attribute representation, preferred location (register, stack, heap)

Implementation of elementary data types

Involves:

- Storage representation for data objects
- Set of algorithms or procedures for creating and manipulating data objects on their storage representation.

Common Data Types

- Boolean

Boolean Data Type in programming

Propositional (logical) connective	In Programming
and	and, &&
or	or ,
not	!

Boolean Data Type

- A data object can have one of two values – True, False.
 - $S = \{T, F\}$
- Common Operations (Algebra of Booleans)
 - $\text{and} : S \times S \rightarrow S$
 - $\text{or} : S \times S \rightarrow S$
 - $\text{not} : S \rightarrow S$

Boolean Data Type - Value Representation

- A single bit in the storage could be used to store a boolean data value.
- However single bits may not be separately addressable in memory.
- Therefore often a single addressable unit in memory such as a byte or word is used to represent a boolean value.
- A particular bit, typically the sign bit, is used for the value.
- A zero value in the entire storage unit represented false and any other non-zero value represented true.
- Since a large amount of storage is necessary for these representation, provision is often made in some languages for collection of bits.

Common Data Types

- Character

Character Type

- A character data object has a single character as its value.
- Stored in computers as numeric codes.
- Most commonly used coding
 - ASCII (American Standard Code for Information Interchange)
 - Uses 0..127 to code 128 different characters
 - Unicode
 - 16-bit character set.
 - Java is the first widely used language to use the Unicode character set

Character Type

- A **character set** is the aggregate of characters that are used in a computer system or programming language.
- In a coded character set, each character is assigned to a different number within a specific range (Ex: 0 to 255).
- The numbers are called **code points**.
- The assignments of numbers to characters in a set are collectively called a **code page**.
- The ordering of the characters in a character set is called the **collating sequence** for the character set – defines the ordering of characters for relational operations.

Common Data Types

- Numeric Data Type

 - Integers**

 - Floating point real numbers

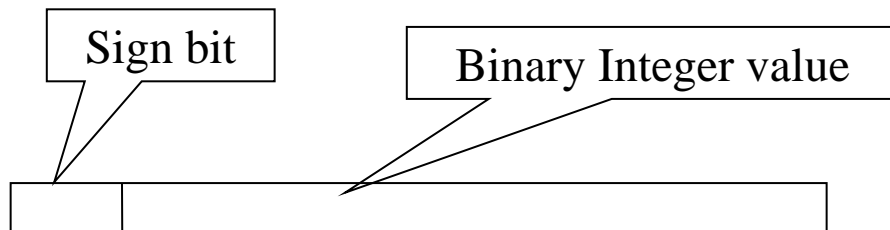
 - Fixed point real numbers

Integer Data Type

- Integer Types are directly supported by the hardware.
- The maximum integer value is sometimes represented as a defined constant.
 - Example: In Pascal - maxint
- Many Computers support several sizes of integers.
 - Example: In C - Short Integer, integer, long Integer

Storage representation of integers

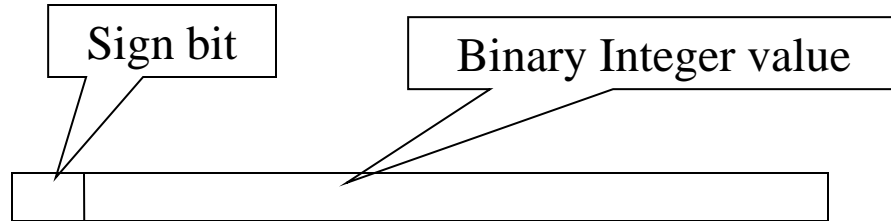
- An integer is represented by a string of bits.
- Typically the leftmost bit represents the sign and the remaining bits represent the absolute value of the number ; **Sign-Magnitude Notation**



- No run-time descriptor
- Possible when the language provides declarations and static type checking.

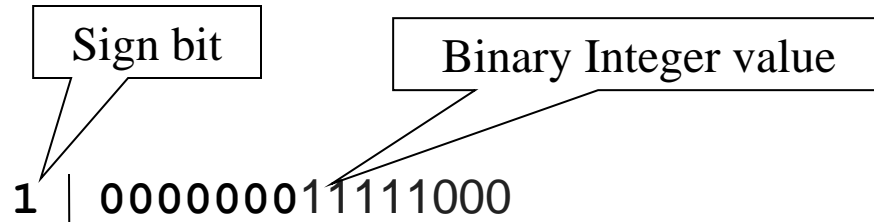
Storage representation of integers

- Convert $(-)$ 248 into Sign-Magnitude format assuming 16 bits
 - For $-$, sign bit is 1. For $+$ sign bit is 0



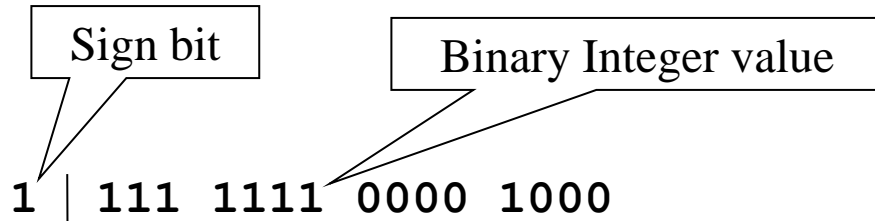
Storage representation of integers

- Convert (-)248 into Sign-Magnitude format assuming 16 bits
 - For -, sign bit is 1. For + sign bit is 0



Storage representation of integers

- Convert (-)248 into 2's complement format
 - 248 → 0000 0000 1111 1000
 - Invert → 1111 1111 0000 0111
 - 2's complement → $1111\ 1111\ 0000\ 0111 + 1 = 1111\ 1111\ 0000\ 1000$



Operations on integer data

Include the following main groups.

- Arithmetic operations
- Binary operations with the signature
 - $\text{BinOp} : \text{integer} \times \text{integer} \rightarrow \text{integer}$
- Unary operations
 - $\text{UnaryOp} : \text{integer} \rightarrow \text{integer}$
- Relation Operators
 - $\text{RelOp} : \text{integer} \times \text{integer} \rightarrow \text{Boolean}$

Operations on integer data (Cont...)

- Assignment
 - Assignment : integer \times integer \rightarrow integer
 - Set the value contained in the first object to be a copy of the value contained in the second object and return an integer value
 - Assignment : integer \times integer \rightarrow void
- Bit Operations
 - BitOp : integer \times integer \rightarrow integer

Common Data Types

-Numeric Data Type

Integers

Floating point real numbers

Fixed point real numbers

Floating Point Data Type

Examples:

Real – FORTRAN

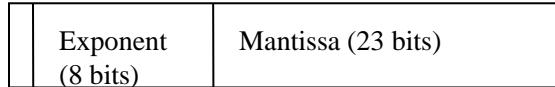
Float - C

Floating point data types model real numbers

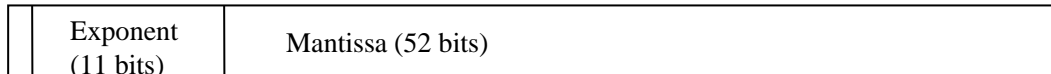
However the representations are only approximations for most real values.

Floating Point Data Type

- Floating-point values are represented as fractions and exponents, called the floating-point representation :
 - Most machines use the IEEE Floating-point standard 754 format.
- Single precision (32 bit) and double precision (64 bits) floating point variables



Sign bit

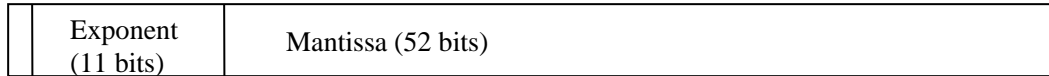


Floating Point Data Type

- Convert -248.75 to the IEEE Floating-point standard 754 - 32 bits
- For single precision bias is 127. Exponent = bias + e



Sign bit



Floating Point Data Type

- Convert -248.75 to the IEEE Floating-point standard 754 - 32 bits
- For single precision bias is 127
 - $248.75 \rightarrow 1111\ 1000.11 = 1.111100011 \times 2^7$
 - **Mantissa** $\Rightarrow 1111\ 0001\ 1000\ 0000\ 0000\ 000$
 - **Exponent** $\Rightarrow \text{bias} + 7 \rightarrow 0111\ 1111 + 0000\ 0111 = 1000\ 0110$

1	1	1	1	1	0	0	0	.	1	1
2^7	2^6	2^5	2^4		2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

	Exponent (8 bits)	Mantissa (23 bits)
--	----------------------	--------------------

1 | 1000 0110 | 1111 0001 1000 0000 0000 000

Floating Point Data Type

Language Implementers use whatever representation is supported by the hardware.

Some smaller computers do not have hardware support for floating-point operations. They emulate these operations in software, which can be 10 to 100 times slower than their hardware implementations.

The same operations described for integers are usually provided for reals. In addition additional operations such as sin, cos, tan may be provided.

Due to round-off issues, equality between two real numbers is rarely achieved.

Boolean operations are restricted.

Common Data Types

-Numeric Data Type

Integers

Floating point real numbers

Fixed point real numbers

Decimal Type - Fixed-Point Real Numbers

Decimal data types store a fixed number of decimal digits, with the **decimal point at a fixed position in the value.**

Example in COBOL Fixed-point reals are declared as below

```
X PICTURE 999V99
```

Most large computers that are designed to support business systems applications have hardware support for decimal data types.

In some cases may be simulated by software.

Decimal Type - Fixed-Point Real Numbers

Have the advantage of being capable of precisely storing decimal values, which cannot be done in floating-point.

Data are stored

- Like character strings, using binary codes for the decimal digits – called binary coded decimal (BCD)
- As integers, with the decimal point being an attribute of the data, typically called scale factor.
 - Decimal point is not stored with the value
 - Data is stored as integers, with the decimal point being an attribute of the data object

Decimal	Binay (BCD)			
	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Decimal Type - Fixed-Point Real Numbers

Have the advantage of being capable of precisely storing decimal values, which cannot be done in floating-point.

Data are stored

- Like character strings, using binary codes for the decimal digits – called binary coded decimal (BCD)
- As integers, with the decimal point being an attribute of the data, typically called scale factor.
 - Decimal point is not stored with the value
 - Since scale factors of variables are known at compile time, the translator can generate appropriate target code.

Decimal Type - Fixed-Point Real Numbers

Example : In PL/I, fixed data are of type Fixed Decimal.

DECLARE X FIXED DECIMAL (10,3),

Y FIXED DECIMAL (10,2),

Z FIXED DECIMAL (10,3);

Let $X = 100.421$, $Y = 34.34$, and Z has a scale factor 3.

$Z = X + Y$?

$z = 100.421 + 34.34$

$= 134.761$

$Z = (X + 10^*Y)$

1	0	0	4	2	1
0	3	4	3	4	0
1	3	4	7	6	1

Decimal Type - Fixed-Point Real Numbers

Stored either one digit per byte or two digits per byte.

- Takes more spaces than the corresponding binary representation.
- Example: 24 bits are required to store a six-digit in BCD and only requires 20 bits in binary.
- Disadvantages
 - Range of values is restricted because no exponents are allowed.
 - Representation take more memory spaces than floating-point and binary types.

Decimal Type - Fixed-Point Real Numbers

Fixed point real numbers may not be an appropriate representation of numbers for some scientific and technological applications.

Why ?

- Representation of $14/1000$ and $6/1000$ by using fixed point representation with two decimal places would be 0.01.
- However the first number is more than twice in magnitude.
- Real numbers that are very small are not clearly distinguishable in fixed point real number representation.

Common Data Types

- Character string type

Character String Type

- Values consist of sequence of characters.
- Design Issues
 - Should strings be a special kind of character array or a primitive type ?
 - Pascal, C, C++, Ada – array of single characters
 - FORTRAN 77, BASIC – primitive type
 - Should strings have static or dynamic length ?
 - Some languages use the convention that character strings are terminated with a special character, null.
 - This is an alternative to maintaining the length of string variables.

Common string operations

- **String moving** (e.g. strcpy in c)
- **String concatenation** (e.g. strcat in c)
- **Comparing lexicographically**, by the order of their codes
- Returns the number of characters in the string; string **length**.
- **Sub-string selection** using positioning subscripts
- **Pattern matching**: Perl, Python, Ruby has elaborate pattern-matching operations built into the language.
- **Input-output formatting**

String Length : design choices

- **Static length string** : length is static and specified in the declaration. Static length string are always full; if a string is shorter, the remaining places are usually filled with blank characters.
- **Limited dynamic length strings** : strings have varying length up to a declared and fixed maximum set by the variable's declaration.
- **Dynamic length strings** : strings have varying length with no maximum.

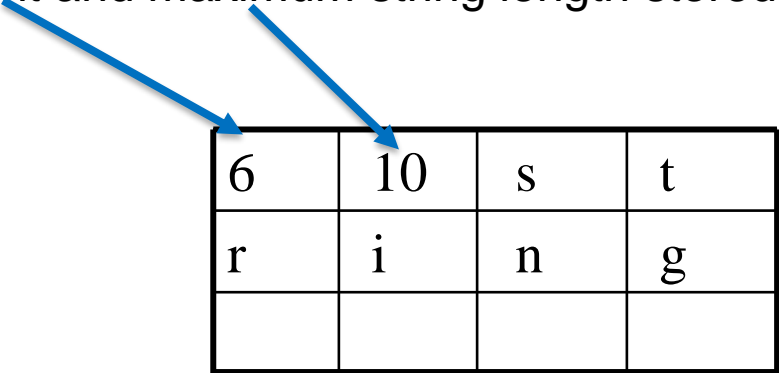
Fixed declared length

s	t	r	i
n	g		

Strings stored 4 characters per word padded with blanks

Variable length with a bound

Current and maximum string length stored at header of string

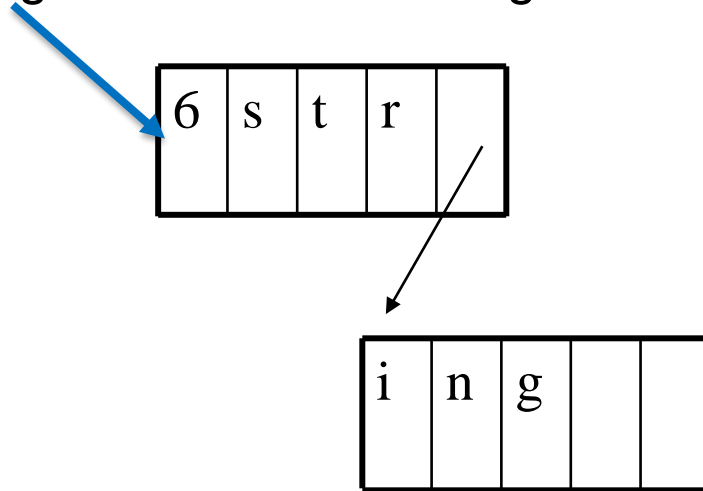


The diagram illustrates a string header structure. A text label 'Current and maximum string length stored at header of string' has two blue arrows pointing to the first two columns of a table. The table is a 3x4 grid. The first row contains the values '6', '10', 's', and 't'. The second row contains 'r', 'i', 'n', and 'g'. The third row is empty. The first two columns represent the current and maximum string lengths, while the last two columns represent the string characters.

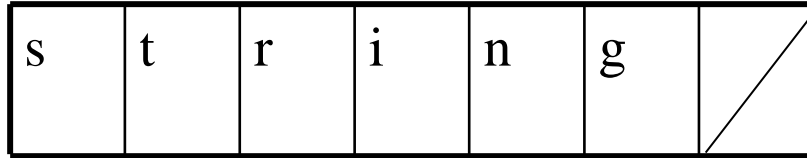
6	10	s	t
r	i	n	g

Unbound with fixed allocation

Strings stored at 4 characters per block.
Length at header of string



Unbound with Variable allocation



String stored as contiguous Array of characters.
Terminated by null character

Implementation

Typically, a **static** character string variable has three fields

- A name, Address of the first character, Length (in characters)

Limited dynamic strings required to store

- A name, Address of the first character, Fixed maximum length, Current length

Dynamic length strings require only

- A name, Address of the first character, The current length to be stored

Implementation

In C and C++ dynamic strings

- The end of a string is marked with the null character.
- Thus it is not necessary to store the current length.
- Not necessary to store maximum length, because index values in array references are not range-checked.

Implementation

There are two possible approaches to dynamically allocate storage for dynamic length strings

- **Storing strings in linked lists**

- Disadvantage : large amount of storage should be allocated to maintain links

- **Store complete strings in adjacent storage cells.**

- When a string grows, and if the adjacent storage is not available, a new area of memory big enough to store the new string is acquired, the full string is copied to this new location. Finally the memory cells used for the old string are deallocated.

Regular Expressions

- Regular expressions describe text patterns.
- Regular expressions are comprised of *characters and meta-characters* and have their own syntax.
- Different versions of regular expression syntax are been used today.
- Abbreviations – regexp, regex

Regular Expressions ...

- Regular expressions describe patterns. However matching text with patterns is carried-out by a regular expression engine.
- Regular expressions does not consider the text semantics, instead the entire text is treated just as a sequence of characters.

Example :

The pattern '*cat*' match with the following texts.

- *My cat*
- *catalogue*

- Generally, a regular expression engine stops the matching process after it detects the first match unless otherwise specified.

Recursive Definition of a Regular Expression

- If a and b are regular expressions so are,
 - $a \mid b$ choice
 - ab sequence
 - (a) grouping
 - a^* zero or more repetitions

The basic elements of regular expressions

- Character sets
- Quantifiers
- Position indicators

Regular Expressions ...

- Character Classes – [...]

- `[xy]` x or y
- `[a-z]` a or b orz
- `[1-9a-zA-Z]` a or b orz or A or B orZ or 1 or 2 or9
- `[^1-9]` any character other than a digit (negation)

- `.` Any character
Period inside character class loses this special meaning.

Regular Expressions - Character Class Abbreviations

- The following are character class abbreviations are supported by many languages such as .NET, PHP, PERL, RUBY, PYTHON, Javascript .

<code>\d</code>	- Match any character in the range 0 – 9
<code>\D</code>	- Match any character NOT in the range 0 - 9
<code>\s</code>	- Match any whitespace characters (space, tab etc.)
<code>\S</code>	- Match any character NOT whitespace (space, tab)
<code>\w</code>	- Match any character in the range 0 - 9, A - Z and a – z
<code>\W</code>	- Match any character NOT the range 0 - 9, A - Z and a - z

Regular Expressions - Quantifiers

- Specify how often a character, character class may or must be repeated in sequence.

General form {min,max}

- Generally quantifiers apply to the item that precede it.
- To figure out the number of a particular backreference, scan the regular expression from left to right and count the opening round brackets. The first bracket starts backreference number one, the second number two etc.

Quantifiers	Meaning
*	0 or more
+	1 or more
?	0 or 1
{3}	Exactly 3
{3,}	3 or more
{3,5}	3,4,or 5

Backreference

\x	Group/subpattern number "x"
----	-----------------------------

Regular Expressions ...

- By using round brackets parts of a regular expression can be grouped together.
- Besides grouping, round brackets also create a "backreference".
- Backreferences allow matching parts to be re-used.
- Back-referenced parts can be accessed by using the notation `\1,\2,...` etc

Regular Expressions - Examples

Examples

- `xy{1,2}z` matches with `xyz` and `xyyz`
- `[aeiou]{0,3}` any sequence of 0 to 3 vowels
- `-{5,5}` exactly 5 dashes
- `T{1,}` one or more Ts. This is the same as `T+`
- `([a-c])x\1x\1` will match `axaxa`, `bxbxb` and `cxcxc`

Regular Expressions ...

- **Anchors** : Meta characters that force the search to start relative to some specific point in the text string.

<code>^</code>	Start of the string.
<code>\$</code>	End of the string
<code>\Z</code>	End of string
<code>\b</code>	Word boundary
<code>\B</code>	Not word boundary
<code>\<</code>	Start of word
<code>\></code>	End of word

<https://cheatography.com/davechild/cheat-sheets/regular-expressions/pdf/>

Regular Expressions - Examples

Consider the following word list

cat

caterpillar

catalogue

fat cat

egrep '^cat' - all except the last

egrep 'cat\$' - first and the last

Regular Expressions - Examples

PHP pattern matching and replacing- preg_replace command

preg_replace (\$pattern , \$replacement , \$subject)

a)

```
$s = 'L e   cture'
```

```
$s1 = preg_replace('/\s+/', '', $s);
```

```
$s1 = 'Lecture'
```

\s

Match any whitespace characters
(space, tab etc.)

b)

```
$s = "L ec   ture ";
```

```
$s1 = preg_replace('/\W+/', '', $s);
```

```
$s1 = "Lecture"
```

\W

Match any character NOT the range
0 - 9, A - Z and a - z

Ordinal Data Types

- **Enumeration**
- Sub-range

Enumeration Type

An enumeration is an ordered list of distinct values.

The programmer defines both the literal names to be used for the values and their ordering in a declaration.

The values in the definition become symbolic constants.

Example:

In Ada

```
Type Days in (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Basic operations

- Relational operations (equal, less-than)

Example :

```
var day var: Days;  
if dayvar > Fri
```

- Assignment
- Operations such as successor and predecessor
Successor : next value
Predecessor : previous value.
- Language may allow enumerated type variables to be used as loop variables or as array subscripts.

Example:

```
var dayvar: Days;  
for dayvar in Mon..Fri loop
```

Design Issues

Type Days in (Mon,Tue,Wed, Thu, Fri, Sat,Sun);

Type holyDays in (Sat,Sun);

.....

X = Sun;

.....

What should be the correct type of 'Sun'?

Enumeration Data Type - Design Issues

Is a literal constant allowed to appear in more than one type definition.

If so, how is the type of an occurrence of such a literal is checked in the program.

- In Pascal a literal constant is not allowed to be used in more than one enumeration type definition.
- In Ada a literal is allowed to be used in more than one enumeration definition – **overloaded literals**.
- The type of such a variable should be determinable from the context of its appearance.

Enumeration Data Type - Advantages

Enumeration types improve both the readability and reliability.

- Named values are easily recognized than coded values.
- Range errors could easily be detected.

Ordinal Data Types

- Enumeration
- **Sub-range**

Sub-range Types

- Sub-range types are not new types. Contiguous subsequence (a part of) of an ordinal type
- They define new names for possibly restricted or constrained versions of existing types.
- All the operations defined for the parent type are also defined for the subrange type, except assignment of values outside the specified range.

Example : in Pascal

Type uppercase = 'A' .. 'Z';

- Subrange of **Character** type

Type myinteger = 1..20;

- Subrange of **Integer** type

Example : in Ada

type Days is (mon, tue, wed, thu, fri, sat, sun);

subtype Weekdays is Days range mon..fri;

- Subrange of **Days** type

Sub-range Types

In some languages sub-range types are inbuilt to the language.

C doesn't support subrange types like Pascal does, with automatic range checking.

Eg: Pascal, Ada

- In C int, long, short and char are all variations of integer data type.
- C char type inherits the operations of integer type.

Subrange Types - Implementation

Usually implemented by associating a non-negative integer value with each symbolic constant in the type.

Range checking must be implicitly included by the compiler in every assignment of a variable or expression to a sub-range variable

Array (Vector) Data Type

Array Data Type

One-dimensional arrays - Vector

Two-dimensional arrays - Matrix

Multi-dimensional arrays

Array Data Type

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

Individual data elements of an array are of some previously defined type.

PHP associative arrays?

```
$age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");
```

Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the **aggregate name**, and the second part is a possibly ***dynamic selector*** consisting of one or more items known as **subscripts or indexes**. Such references require a run-time calculation to determine the memory location being referenced.

Array Data Type - Design Issues

- When does array *allocation* take place ?
- Can arrays be *initialized* when they have their storage allocated ?
- What types are legal for *subscripts* ?
 - Often a subrange of integers
- What is the **lower bound of the subscript** range and is it implicit ?
 - In C, C++ and Java Lower bound is 0, in FORTRAN 77 *default* is 1
 - In most languages lower bound can be specified by the programmer
- **Number of array subscripts** (one, two, three,...)
 - C can have only one subscript, but arrays can have arrays as elements.
- Are subscripting expressions in element references **range checked** ?
 - C, C++ and FORTRAN do not specify range checking.
 - Pascal, Ada and Java perform range checking.
- Problems in using parentheses to enclose array subscripts
 - Example: B(I) - Is this a function call or an array reference ?

Types of Arrays - Static arrays

Subscript ranges are statically bound
and storage *allocation* is static (done before run time)

- **Advantages** : Efficiency
- **Disadvantages**: Size must be known at compile time.
Bindings are fixed for entire program.
- **Example** :

Global arrays in C, static arrays (C/C++)

- The *subscript type* (integers) is bound at the language design time
- The *subscript ranges* are statically bound
- *Storage* is statically allocated

Types of Arrays - Fixed-dynamic arrays

Subscript ranges are statically bound,

But the storage *allocation* is done at declaration elaboration time during execution

- **Advantages** : Space efficiency; same memory locations can be used for other purposes
- **Disadvantages**: Size must be known at compile time.
- **Example**:
Most Java locals, and C locals that are not static.

Types of Arrays - Stack-dynamic arrays

Subscript ranges are dynamically bound and, storage *allocation* is done dynamically.

However once this is done, they remain fixed during the lifetime of the variable.

- **Advantages** : flexibility
- **Disadvantage**: once created, array size is fixed

Types of Arrays - Heap-dynamic arrays

Binding of *subscript ranges* and storage *allocation* is dynamic in heap
Can change at any time during the array's lifetime.

- **Advantages** : flexibility
- **Example** VB arrays

In APL, Perl and JavaScript, arrays grow and shrink as needed

C and C++ allow heap-dynamic arrays using pointers

In Java, all arrays are objects (heap dynamic)

C# provides both heap-dynamic and fixed-heap dynamic

Summary of Types of Array

	Subscript range	Storage
Static	Compile time	Compile time
Fixed dynamic	Compile time	Declaration elaboration time
Stack dynamic	Runtime but fixed thereafter	Runtime but fixed thereafter
Heap-dynamic	Runtime	Runtime

Array Data Type - Initialization

FORTRAN77, ANSI C and C++ allow array initialization in the declaration. Can let the compiler count them

- In C: `int list[] = {5,7,10,12}`

Pascal and Modula-2 do not allow array initialization in the declaration section

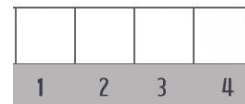
Array Data Type - Implementation

- Implementation arrays requires more compile-time efforts than does implementing simple types.
- *The code to access array elements must be generated at compile time.*
- A single-dimension array elements can be accessed by using the function, when the first element is given the subscript 1.

`address(list[k])`

`= address(list[1]) + (k-1)* element_size`

`= address(list[1])-element_size + k * element_size`



The first part is a **constant**, whereas the second part is a **variable**

- If the element type is statically bound and the array is statically bound to storage, the value of the constant part can be computed before run time.

Array Data Type - Implementation

The descriptor stores information required to construct access information

If run-time checking of index ranges is not done and the attributes are all static,

- then only the access function is required at run time;
- no descriptor is required.

Array
Element type
Index Type
Index lower bound
Index upper bound
Address

Compile-time descriptor
for single dimensional arrays

Array Data Type - Implementation

Hardware memory is linear. Thus values of data types that have **two or more dimensions must be mapped onto a single dimensioned memory.**

Two common ways to map multidimensional arrays to one dimension

- Row major order
- Column major order

Example:

3	4	7
6	2	5
1	3	8

Row major order - 3, 4, 7, 6, 2, 5, 1, 3, 8

Column major order - 3, 6, 1, 4, 2, 3, 7, 5, 8

Record Data Type

Records

A record is a possibly **heterogeneous aggregate of data elements** in which the individual elements are identified by **identifiers**.

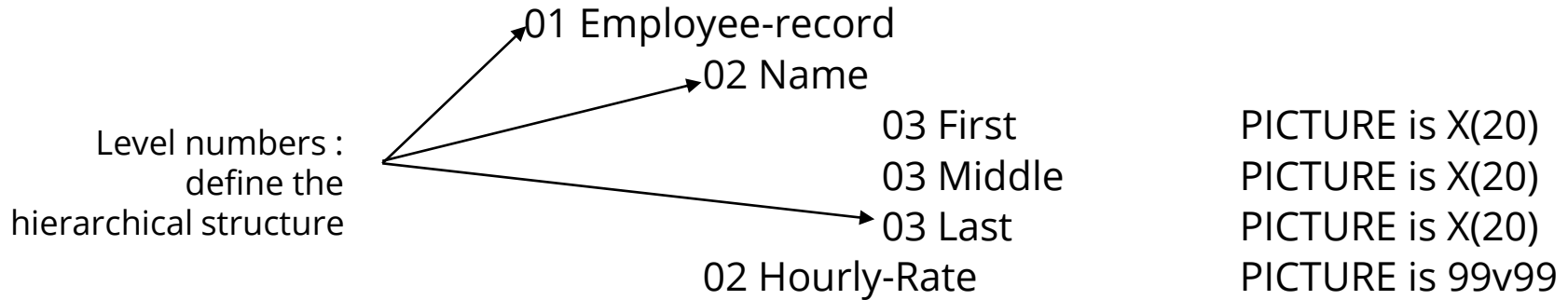
Design issues

- What is the syntactic form of references to fields ?
- Are elliptical references allowed ?

Definition of records

A number of different ways

- Example COBOL Record Definition



Format of the field locations

Definition of records

Example in Pascal, Ada definition of records

```
Employee_record :  
record  
    name :  
        record  
            first : string(1..20);  
            middle : string(1..10);  
            last : string(1..20);  
        end record;  
    hourly_rate : float;  
end record;
```

Reference to Fields

Fully qualified reference to a record field

- one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.
- In COBOL :
 - field_name **OF** record_name_1 **OF...OF** record_name_n
 - middle **OF** Name **OF** Employee-record
- Most other languages use dot notation
 - Employee_record.Name.Middle

Reference to Fields

- **Elliptical reference**

- in an elliptical reference the field is named by *omitting enclosing record names*.
- This is allowed as long as the resulting reference is unambiguous in the referencing environment.

Example: In Pascal **with** keyword can be used to create elliptical references

- Record:

```
TheCustomer.Name := 'Michael';
```

```
TheCustomer.Flight := 'PS901';
```

- Reference:

```
With TheCustomer do
```

```
begin
```

```
    Name := 'Michael';
```

```
    Flight := 'PS901';
```

```
end;
```

Reference to Fields : Elliptical references

Example: In COBOL

- First
- First of Name
- First of Employee-Record are elliptical references.

01 Employee-record

02 Name

03 First

PICTURE is X(20)

03 Middle

PICTURE is X(20)

03 Last

PICTURE is X(20)

02 Hourly-Rate

PICTURE is 99v99

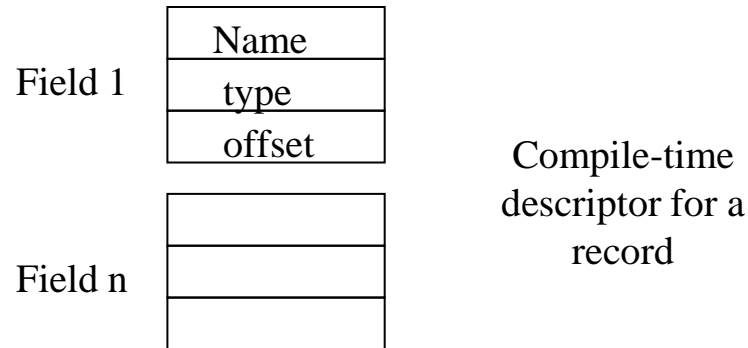
Reference to Fields : Operations

- Assignment Operation
- Record Comparison
- Record Initialization

Reference to Fields : Implementation

The fields of records are stored **in adjacent memory locations**.

Field accesses are all handled using offsets, relative to the beginning of the record.



Files

Files

A file is a data structure with the following properties

- Consists of records.
- Typically stored *in a secondary storage*.
 - Size may be much *larger* than a data object of other types.
- *Lifespan* extends beyond the lifetime of the program that creates it.

Files

General Uses

- For input/output data to external operating environment.
- Temporary scratch for data when not enough high-speed memory is available.

The types of data that cannot be stored in files.

- Variable types of records in a single file.
- Data structures with pointer data objects.

Files - Types

Different types of files based on how data stored in a file can be **accessed**)

- **Sequential files**
- **Direct Access Files**
- **Indexed Sequential Files**

Files

Sequential files

- Records can be accessed sequentially in the order in which they appear in the file.
- Records can be added at the end of the file

Files - Types

Direct Access files

- Every record is *associated with a key*.
- Any record can be accessed at random.
- Data are *stored as a set of unorganized records*
- The record to be accessed is specified by using a key.
- An *index (key, location pairs)* is associated with the file to facilitate random access of records

Files - Types

Indexed sequential files.

- Similar to direct access file.
- Index is associated not generally with each record (**Sparse Index**), but for a group of records, and points to the first record in the group.
- Records also can be accessed in sequence, beginning from an index selected at random.
- Accessing records in an indexed sequential file.
 - If key is in index, location of the record is known.
 - Otherwise, the closest index indicates the group where the record should be found.
 - Record can be located by a sequential search from the start of record group to end of the group.

Files - Operations

- **Open** : Specify the name of the file and the data access mode (read/write)
- **Read** : Transfers the content of the current file component to a designated program variable.
- **Write** : Creates a new file component and transfers the contents of a designated program variable to this file component.
- **end-of-file test** : Checks whether the read operation has reached the last component of the file.
- **Close** : Indicates to the OS to unlink the file from the program.

Pointers

Pointer Types

Defines a class of data objects whose values are locations of other data objects

Anonymous variables: variables without names

Pointer variables are typically used to reference some other variable, rather than being used to store data of some sort.

Pointer Types

Pointers have been designed for 2 uses

- To provide power of indirect addressing
- To provide a method of dynamic storage management

Pointer Types - Design Issues

- What are the **scope and lifetime** of a pointer 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, reference types (e.g. Java), or both?

L-values and R-values

Every memory cell has

- Address (unique) sometimes called **l-value**
- Value sometimes called **r-value**

For example in the assignment statement $x = y + 1$

we are interested in the **l-value** of x and **r-value** of y .

Constants, like 1, only have **r-values**.

Pointer Types

To support the pointer data type a language should provide the following features.

- An elementary data type pointer
 - where the value of a data object of this type is a location of another data object.
- A creation operator which is capable of
 - Allocate a block of storage of fixed size to the new object and returns its l-value.
 - Create a pointer variable and store the l-value returned in the previous step as its r-value.
- A dereferencing operation
 - to access the data object pointed by the pointer variable.

Pointer Types

Data object in pointer type might be treated in two ways

Pointers may refer data object of a single type.

The pointer data object may be restricted to point only to data objects of the same type.

Example : C, Pascal

Static type checking is possible

Pointers may refer data objects of any type.

Data objects carry type descriptors during execution.

Example : Smalltalk

Dynamic type checking has to be performed

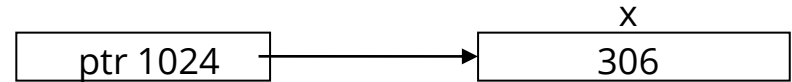
Pointer Types - Operations

Assignment :

- Sets a pointer variable's value to some address
- A predefined value nil(or null) may use to indicates that a pointer cannot currently be used to reference any memory cell.

Dereferencing :

- Considering the value of a pointer variable as an address of a another memory cell, and retrieving the value of that memory cell.
 - A normal reference to ptr yields 1024
 - A dereferenced reference to ptr yields 306
- Dereferencing can be either:
 - Implicit e.g. ALGOL 68, FORTRAN 90
 - Explicit e.g. Pascal, c
 - Pascal uses the ^ postfix unary operator
 - C uses * or -> for dereferencing



Pointer Types - Implementation

Pointer data object represents a storage location which contains the address of another storage location.

This address may be the *base address* of a block of storage of a collection of data objects (eg. *pointer to a data structure*).

Two major storage addressing methods can be used when creating pointer variables.

Absolute addresses

- A pointer represents an actual memory address in the storage.

Relative addresses

- A pointer represents an offset from the base address of some heap storage block within which the data object is allocated storage.

Pointer Types - Implementation

Absolute addresses - Advantages & Disadvantages

- Storage may allocate from anywhere in memory.
- Memory access is efficient (Direct Addressing)
- Storage management is difficult.
 - Data objects cannot be moved to a different location.
- Garbage collection is difficult.
 - Storage allocated for each such object must be recovered individually.

Pointer Types - Implementation

Relative addresses - Advantages & Disadvantages

- Need to allocate a block of storage initially,
 - Within which memory is allocated for pointer variables subsequently.
- A separate data block may be allocated for each separate data type.
 - Storage management is simple.
- Memory access is more costly. Base address + offset
- Data block can be moved to a different location at any time without invalidating the corresponding pointers.
- Storage required for a block may be acquired on entry to a program and can be deleted on exit.

Pointers - Problems

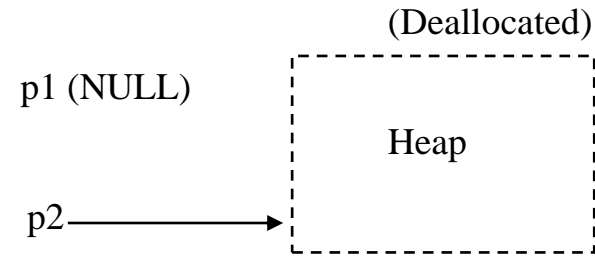
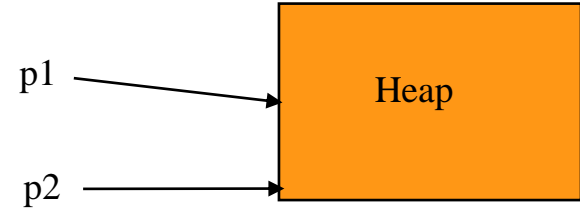
Dangling Pointer or dangling reference

A pointer that contains the address of a heap-dynamic variable that has been deallocated

How a dangling pointer is created

- Create a new pointer variable p1
- p2 = p1
- Deallocate p1
- Now p2 is dangling

Dangling pointers may destroy required/ important data



Garbage?

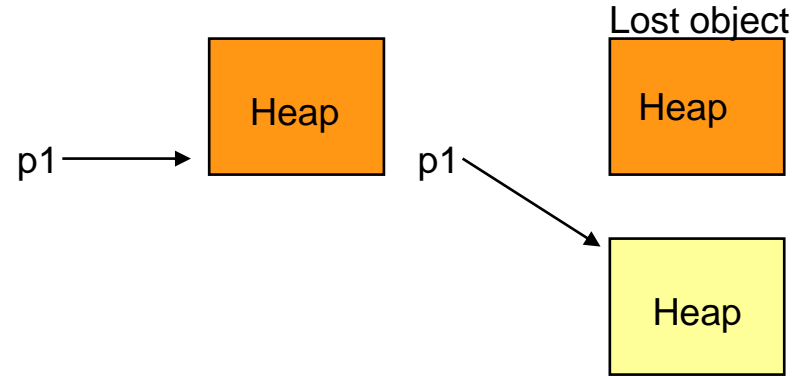
Lost Heap-dynamic variables or Garbage :

An allocated heap-dynamic variable that is no longer accessible to the user program

- Cannot be used for their original purpose
- Cannot be allocated for some useful work

How garbage is created

- Create a pointer variable p1
- ...
- $p1 = p2$
- Now the original memory allocated to p1 cannot be accessed



Languages that require explicit deallocation of dynamic variables suffer this problem
– **memory leakage**

Object Destruction

- Explicit
 - Example `dispose(x);`
- Implicit – Automatic Garbage Collection
 - In Java if no live references exists for an object it is garbage
 - In JavaScript a hint can be given to the browser by assigning the 'null' value to a variable, if the value of a variable is no longer needed.

Type Checking

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

Type errors is the application of an operator to an operand of an inappropriate type.

A **Compatible type** is one that is either

- Legal for the operator or
- Allowed under language rules to be implicitly converted by compiler-generated code to a legal type

Type Checking

Conventional computer hardware generally unreliable in detecting data type errors.

During program translation (compilation) information on data objects are gathered into symbol table (data object attributes). In subsequent steps, each operation invoked is checked for the validity of types of arguments.

If during type checking, a mismatch occurs between the actual type of an argument and the expected type for that operation, then either

- The type mismatch may be **flagged as an error** and an appropriate error action taken or
- A **coercion** may be applied to change the type of the actual argument to the correct type.

Type Conversion

Languages provide **type conversion** in two ways.

- As a set of built-in functions that the programmer may **explicitly** invoke.
- Automatic conversion in certain cases of type mismatch : **Coercion**.
 - The basic principle of coercions is **not to lose information**.

Example:

short int -> long int -> real - is called widening or promotion

The reverse operation is called narrowing. PL/I's conversion operations permit narrowing coercions.

Type Compatibility

Two different types of type compatibility methods

- Name type compatibility
 - Type names will be compared
 - In Pascal Name equivalence is required in parameter passing
- Structure type compatibility
 - Structure of the types will be compared
 - Structural equivalence is supported in FORTRAN and ALGOL.

Type Compatibility- Name Type Compatibility(Nominative type system)

Only the type names must be compared to determine compatibility.

Two variables have compatible types only when they are either

- In the same declaration
- In declarations that use the same type name.

Example: In Pascal

```
type indextype = 1.. 100 { a sub range type}  
var count : integer;  
    index : indextype;
```

The variables **count** and **index** are not compatible.
Count could not be assigned to index or vice versa.

Type Compatibility- structure type compatibility (Structural type system)

Two data objects are considered to be of compatible types

- If there types have identical structures.

For each element within a type there must be a corresponding and identical feature in other type.

The entire structures of the two types must be compared to determine compatibility.

Type Compatibility - example

Which variables (a,b,c,d) are type compatible?

Type

month = 1..12

hour = 1..12

array1 = array[1..12] of integer;

array2 = array[month] of integer;

array3 = array[hour] of integer;

var

a : array1;

b : array2;

c,d : array3;

Under **name compatibility** c and d are compatible but a,b,c are not.

Under **structure compatibility** they are compatible.

- Is it sensible to intermix items of array2 with array3?
- Structurally equivalent objects may be semantically different.
- Less type safe

Strong Typing

If all type errors in a program can be detected statically then the language is said to be strongly typed.

A function f with the signature $f: S \rightarrow R$ is called type safe if execution of f cannot generate a value outside of R .

- If a operation is type safe **no dynamic type checking is needed** as the results are of correct type
- True strong typing is generally difficult : type conversion facilities

Strong Typing

Example: (In C)

Is $*$ in C is type safe ?

If x and y are of type short integers $x * y$ may generate values outside the range allowed for short integers.

Therefore $*$ is not type safe.