Chapter 6

Data Types

"Computer programs produces results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types matches the real-world problem space"

6.1 Introduction

- **Data Type defines**
  - a collection of data values
  - a set of predefined operations on those values
    - ❖ what operations are defined and how are they specified?

- **How well the data types match the real-world problem space is the important factor in determining the easy of programming**
  - *pre-90 FORTRAN*
    - ⇔ all problem space data structures had to be modeled with only a few basic language-supported data structures
    - ⇔ in pre-90 FORTRANs, linked lists, nonlinked list, and binary trees are all commonly implemented with arrays (no record type)
  - *COBOL*
    - ⇔ decimal data values, a structured data type for record
  - *PL/I*
    - ⇔ many data types, with the intent of supporting a large range of applications
  - *ALGOL-68*
    - ⇔ provides a few basic data types and a flexible combining methods that allow programmer to tailor a structure to the problem at hand (*user-defined types*)
  - *Ada*
    - ⇔ users are allowed to create a unique type for each unique class of variables (*abstract data type*)

6.2 Primitive Data Types

- **Primitive data types**
  - *the data type that are not defined in terms of other types*
  - the data types whose representation and operations are closely supported by hardware are called *primitive data types (reflection of hardware)*
  - they are used, along with one or more type constructor, to provide more complex structured types such as arrays and records

(1) Numeric Types

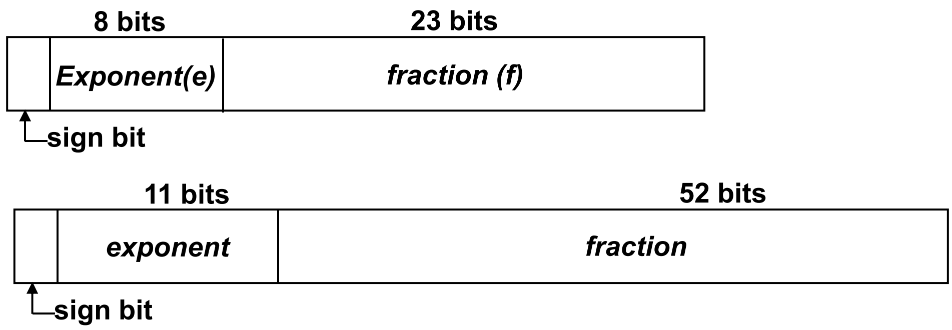
- ***Integer***
  - Most common primitive numeric data type
  - Many computers (CPU Hardware) support several size of integers
    - ⇔ byte, word, long word, quadword
  - These reflected in some programming languages
    - ⇔ Java's signed integer sizes: `byte, short, int, long`
  - Stored in single memory word
    - ⇔ word size ? (16-bit computer, 32-bit computer, …)
  - *Implementation*
    - ⇔ represented by a string of bits, with one of the bits, typically leftmost, representing the sign
    - ⇔ 2's complement

- ***Floating-Point : (fraction + exponents)***
  - Floating-point data types model real numbers, but the representations are only approximations for most real values (how to represent 0.1 in computer precisely ?)
  - Floating-point data types are included in most languages, although many small computers do not have hardware support (FPU) for such types (then how ?)
  - Languages that are designed to support scientific programming generally include two floating-point types ; *float* (stored in single memory word) and *double precision*
  - *Implementation*
    - ⇔ IEEE Floating-Point Standard 754 format
    - ⇔ sign bit is the sign of the fraction
    - ⇔ the exponent is stored in an excess notation

$$f \cdot 2^{(e-bias)}$$

single precision (bias : 127)

double precision (bias 1023)



- ***Decimal***
  - **All mainframe and large minicomputers that are designed to support business systems applications have hardware support for decimal data types**
    - ⇔ **Essential to COBOL**
    - ⇔ **C# offers a decimal data type**
  - **BCD (Binary Coded Decimal)**
  - **Example : "20" ➔ 0000 0010 0000 0000 (unpacked)**
  - **Decimal types have advantage of being capable of precisely storing decimal values, at least those within a restricted range, which can not be done in floating-point**
    - ⇔ ***Advantage*: accuracy**
    - ⇔ ***Disadvantages*: limited range, wastes memory**
  - **Implementation**
    - ⇔ **stored very much like character strings, using binary code (BCD) for the decimal digits**
    - ⇔ **packed, unpacked**
    - ⇔ **decimal add by H/W**

(2) Boolean Types

- **the simplest of all types**
- **their range of values has only two elements, one for true and one for false (for readability)**
- ***Implementation***
  - **can be represented by a single bit, but because a single bit is difficult to access efficiently, typically a byte**

(3) Character Types

- **Characters are stored in computers as numeric codings**
  - **ASCII which uses the values 0..127**
  - **C, C++ : `char` , FORTRAN (`CHARACTER`)**
- ***Implementation***
  - **ASCII (American Standard Code for Information Interchange) : 1 byte**
  - **16-bit Unicode : 2 bytes (UCS-2)**
    - ⇔ **includes the characters from most of the world's natural languages**
    - ⇔ **the first 128 characters of Unicode are identical to those of ASCII**
    - ⇔ **Java, JavaScript, Python, Perl, C#**
  - **32-bit Unicode : 4 bytes (UCS-4)**
    - ⇔ **Supported by Fortran, starting with 2003**

6.3 Character String Types

- A character string type is one in which the objects consists of sequence of characters

(1) Design Issues

- Should strings be a primitive type or simply a special kind of character array ?
- Should string have static or dynamic length ?

(2) Character String Type in Certain Languages

- C and C++
  - Not primitive
  - Use `char` arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions

(3) Primitives or Just a Character Array

- String as a special kind of character array
  - String is usually stored in arrays of single character
  - Pascal, Modula-2, C, Ada, C++

    ⇔ C : a null-terminated character array, <string.h>
- String as a primitive data type
  - String types are important to the writability of a language (and, not so costly)
  - provide assignment, relational operators, catenation, and substring references as a primitive operation
  - FORTRAN77, FORTRAN90, and BASIC

```
CHARACTER BELL*1, C2*2, C3*3, C5*5, C6*6
REAL Z
C2 = 'z'
C3 = 'uvwxyz'
C5 = 'vwxyz'
C5(1:2) = 'AB'
C6 = C5 // C2
I = 'abcd'
Z = 'wxyz'
BELL = CHAR(7)     Control Character (^G)
```

FORTRAN

C2	'z '
C3	'uvw'
C5	'ABxyz'
C6	'ABxyzz '
I	'abcd'
Z	'wxyz'
BELL	07

(3) String Length Options

- *Static length string*
  - the length can be static and specified in the declaration
  - FORTRAN77, FORTRAN90, COBOL, Ada
  - Example
    - ⇔ `CHARACTER(LEN = 15) NAME1, NAME2`
- *Limited dynamic length string*
  - allows strings to have varying length upto a declared and fixed maximum
  - VARYING attribute in PL/1
- *Dynamic length string*
  - allow strings to have varying length with no maximum
  - flexible, but overhead of dynamic storage allocation
  - SNOBOL4

(4) Evaluation

- *String types are important to the writability of a language*
  - Dealing with strings as arrays can be more cumbersome than dealing with a primitive string type
    - ⇔ In C, `strcpy()` → requires loops for assignment
- *The addition of strings as a primitive type to a language is not costly* in terms of either language or compiler complexity
  - String operations such as simple pattern matching and concatenation are essential and should be included for string type values

(5) Implementation

- When character string types are actually character arrays, the language often supplies few operations
- Descriptor for static and dynamic strings

Static String
Length
Address

Compile-time descriptor for static string

Limited Dynamic String
Address
Maximum Length
Current Length

Run-time descriptor for limited dynamic string

** descriptor : the collection of attributes of variable*

- *Two possible approaches to the dynamic allocation*
  1) String can be stored in a linked list (slow, more storage)
  2) To store complete strings in adjacent storage cells
     - ⇔ copy entire string when it grows to available space
     - ⇔ fast referencing, less storage, but slow allocation

6.4 User-Defined Ordinal Types

- An *ordinal type* is one in which the range of possible values can be easily associated with a set of positive integers
  - In Pascal, for example, the primitive ordinal types are `integer`, `char`, and `boolean`
  - `Enumeration` types, `Subrange`

(1) Enumeration Types

- An *enumeration type* is one in which all of the possible values, which are symbolic constants, are enumerated in the definition
  - Example : Ada (C#)

```
type DAYS is (Mon, Tue, Wed, Thu, Fri, Sat, Sun) ;
type WEKEND is (Sat, Sun) ;
int i ;
DAYS a ;
a = Mon ;
```

- Design Issues
  - Is a literal constant allowed to appear in more than one type definitions ?
    - ⇔ overloaded literals
  - If so, how is the type of an occurrence of that literal in the program check ?

- In Pascal (similarly in ANSI C and C++)
  - a literal constant is not allowed to be used in more than one enumeration type definition in a given referencing environment
  - Enumeration type variables can be used as array scripts, for loop variables, and case selector expressions, but can be neither input nor output
  - It can be also compared with the relational operators with their relative positions in the declaration

```
type colortype = (red, blue, green, yellow) ;
var color : color type ;
.....
color := blue ;
if (color > red)  then color := pred(blue) ;
```

Operations
- Predecessor
- Successor
- Position
- Values

- In C,
  - implicitly converted into integer

```
enum day {sun,mon,tue,wed,thu,fri,sat} d1,d2 ;
```

- Why enumeration types ?
  - It provides greater readability in a very direct way
    - ⇔ named values are easily recognized, whereas coded values are not
  - Comparison with numeric types (reliability)
    - ⇔ arithmetic operation is impossible
    - ⇔ range errors can be detected easily

(2) Subrange Types

- a contiguous subsequence of an ordinal type
  - Example in Pascal,

```
type
    uppercase = 'A'..'Z' ;
    index = 1..100 ;
```

- Why ?
  - enhancing the readability and reliability

(3) Implementation

- Enumeration types are usually implemented by associating a nonnegative integer value with each symbol constant in type
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types
- Subrange types are implemented in exactly the same way as their parent types, excepts that range checks must be included in every assignment (efficiency *vs.* safe)

6.5 Array Types

- An *array* is a homogeneous aggregate of data elements in which an individual elements is identified by its position in the aggregate, relative to the first element
  - A reference to an array element in a program often includes one or more non-constant subscript (e.g. `aa[i][j]`)
    - ⇔ such references require additional run-time calculation to determine the memory location being referenced

(1) Design Issues

- What type are legal for subscripts ? (int, ordinal type, …)
- When are subscript ranges bound ? (run-time *vs.* compile-time)
- When does array allocation take place ? (run-time *vs.* compile-time)
- How many subscripts are allowed ? (dimension)
- Can array be initialized when they have their storage allocated ?
- What kind of slices are allowed, if any ?

(2) Arrays and Indices

- Array referencing :
  - *Indexing* (or subscripting) is a mapping from indices to elements
  - two-level syntactic mechanism
    - ⇔ `array_name (index_value_list)` → `an element`
  - aggregate name + index (or subscript) : a[100]
- Selection operation can be thought of as a mapping
  - array_name[index_value_list] -> element
- Parentheses vs. brackets
  - In FORTRAN and PL/1, (*mappings*)
    - ⇔ `SUM = SUM + B(I)`
      - ⇒ How can the compiler distinguish the function call and array indexing ?
  - In Pascal, C, Modular-2,
    - ⇔ `SUM = SUM + B[I] ;`

array type
- element type
- index type

(3) Subscript Binding and Array Categories

- The binding of the subscript type to an array variable is usually static
- Lower bound of subscript range
  - In C, it is zero
  - In FORTRAN I, II, and IV , it is fixed at 1
  - In FORTRAN 77 and 90, default is to 1

int
boolean
enumeration type
subrange type

- Four categories of array can be defined based on the *binding to subscript value ranges* and *binding to storage*
  - *Static Array*
    - ⇔ the subscript value ranges are statically bound and storage allocation is static (done before run time)
    - ⇔ Example : FORTRAN 77 array
    - ⇔ Advantage : execution time efficiency
  - *Fixed statck-dynamic array*
    - ⇔ the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution
    - ⇔ Example :
      - ⇒ In Pascal, local array defined in procedure
    - ⇔ Advantage : space efficiency
  - *Fixed heap-dynamic array*
    - ⇔ the subscript ranges are dynamically bound, and the storage allocation is dynamic (done during run time) but fixed after allocation
    - ⇔ Advantage : flexibility
    - ⇔ Example :
      In Ada (similarly, in C and FORTRAN 90)

```
void *malloc(sizet size)
```

```
GET(LIST_LEN)
declare
  LIST : array (1..LIST_LEN) of INTEGER;
begin
  ...
end ;
```

- *Heap dynamic array*
    - ⇔ the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's life time
    - ⇔ Example : In ALGOL 68,

```
flex [1:3] int list ;
...
list = (3,5,7) ;
list := 67 ;
```

    - ⇔ Advantage : flexibility

- Arrays provided by well-known languages
  - C and C++ arrays that include `static` modifier are static
  - C and C++ arrays without `static` modifier are fixed stack-dynamic
  - C and C++ provide fixed heap-dynamic arrays
  - C# includes a second array class `ArrayList` that provides fixed heap-dynamic
  - Perl, JavaScript, Python, and Ruby support heap-dynamic arrays
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - In Ada, the default is to require range checking, but it can be turned off

(4) Heterogeneous Array

- Heterogeneous array
  - is one which the elements need not be of the same type
  - supported by Perl, Python, JavaScript, and Ruby
    - ⇔ arrays are heap dynamic

(5) The Number of Subscripts in Arrays

- Limitations in the number of array subscripts
  - FORTRAN I : 3
  - FORTRAN 77 and 90 : 7
  - Others : no such limitations
- Array in C can have only one subscripts, but array can have array as elements, thus supporting mutidimensional array
  - Orthogonal design
  - Example

```
int mat[5][4] ;
```

(6) Array Initialization

- In FORTRAN 77, all data storage is statically allocated, so load-time initialization using DATA statement is allowed
  - Example

```
INTEGER LIST(3)
DATA LIST /0, 5, 5/
```

- C
  - also allows initialization of its static array
  - but, not for dynamic array
  - Example

```
int list[] = {4,5,7,83}
char name[] = "Nang Jongho"
char *names [] = {"Mike", "Fred", "Mary Lou"};
```

- Pascal and Modular-2 do not allow array initialization in the declaration of programs
- Ada provides two mechanisms for initializing array :

```
LIST : array (1..5) of INTEGER := (1,3,5,7,9) ;
BUNCH : array (1..5) of INTEGER := (1 =>3, 2=>4, others =>0) ;
```

- Python
  - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

⇨ puts `[0, 9, 36, 81]` in `list`

(7) Array Operations

- Some languages provide operations that deal with array as units

- In FORTRAN 90,
  - It includes a number of array operations that are called *elemental* because they are operations between pairs of array elements
  - The assignment, arithmetic, relational, and logical operators are overloaded for arrays of any size
- In APL,
  - APL is the most powerful array-processing language ever devised
  - the four basic arithmetic operations are defined for vectors and matrices, as well as scalar operand
  - Inner product operator ('.')

⇔ Example :

`A +.* B` → $\sum_{i=0}^{n} A[i] * B[i]$

take other operators as operands

- Python,
  - provides array assignments, but they are only reference changes.
  - also supports array catenation and element membership operations

(8) Slices

- Rectangular and Jagged Arrays
  - A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
  - A jagged matrix has rows with varying number of elements
    - ⇔ Possible when multi-dimensioned arrays actually appear as arrays of arrays
  - Languages
    - ⇔ C, C++, and Java support jagged arrays

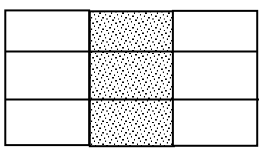
```
char *names [] = {"Mike", "Fred", "Mary Lou"};
```

    - ⇔ Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

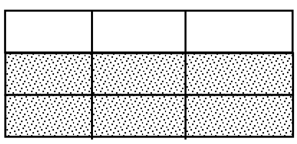
(9) Slices

- A slice of an array is some substructure of that array
  - nothing more than a referencing mechanism
  - a mechanism for referencing part of an array as a unit
  - Example : In FORTRAN 90, (Python)

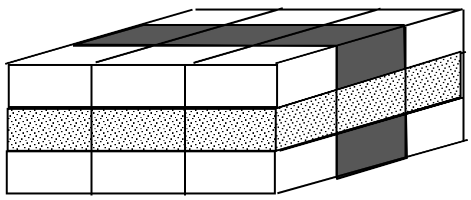
```
INTEGER VECTOR(1:10), MAT(1:3,1:3), CUBE(1:3,1:3,1:3)
MAT = CUBE(1:3,1:3,2)
```



MAT(1:3,2)



MAT(2:3,1:3)



CUBE(2,1:3,1:3)

```
MAT = CUBE(1:3, 1:3, 2)
MAT = CUBE(2, 1:3, 1:3)
```

(10) Implementation

- The code to allow accessing of array elements must be constructed at compile time for efficient run-time access
- Single-dimensioned array

  address(list[k]) = address(list[1])+(k-1) * element_size
  = (<u>address(list[1]) - element_size</u>) + (k*element_size)

  constant, so that can be computed at compile time

Array
Element type
Index type
Index lower bound
Index upper bound

Compile-time descriptor for single-dimensioned array

3	4	7
6	2	5
1	3	8

- raw major order : 3,4,7, 6,2,5, 1,3,8
- column major order : 3,6,1, 4,2,3, 7,5,8

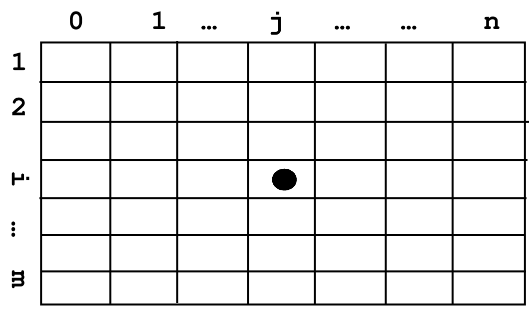
- Mutidimensional Arrays Implementations
  - *Row major order* : the elements of array that have as their first subscript the lower bound value of that subscript are stored first (almost all imperative lang)
  - *Column major order* : the elements of array that have as their last subscript the lower bound value of that subscript are stored first (in FORTRAN), followed by the elements of second value of the last subscript

How about 3-D array ? (a[10][10][10])

- The access function for two-dimensional arrays stored in row major order

  location(a[i,j])
  = (address of a[1,1] + ((((i-1)*n) + (j-1))*element_size)
  = (<u>address of a[1,1]) - ((n+1)*element_size)</u> + ((i*n+j)*element_size)

  constant



The location of the [i,j] element in a matrix

How about the address of a[i][j][k] ?
a[A][B][C]

- A compile-time descriptor for multi-dimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
....
Index range n

6.6 Associative Array

- **Associative Array**
  - an unordered collection of data elements that are indexed by an equal number of values called keys
  - the user defined keys must be stored in the structure
    - ⇔ each element of an associative array is in fact a pair of entities, a key and a value
  - Supporting
    - ⇔ directly by Perl, Python and Ruby
    - ⇔ by standard class libraries of Java, C++, and C#
  - Example (Perl, called ***hash***)

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
$salaries{"Perry"} = 58850;
delete $salaries{"Gary"};
if (exists $salaries{"Shelly"}) …
```

  - a hash is much better than array if searches of the elements are required, because the implicit hashing operation used to access hash elements is very efficient
  - hashes are ideal when the data to be stored is paired
  - if every element of a list must be processed, it is more efficient to use an array
  - Built-in type in Perl, Python, Ruby, and Lua

6.7 Record Types

- A *record* is a *possibly heterogeneous aggregate* of data elements in which the individual elements are *identified by names*
  - It has been a part of all the most popular programming languages, except pre-90 version of FORTRAN, since the early 1960s when they were introduced by COBOL
- Record vs. Array
  - Elements
    - ⇔ Record (heterogeneous), Array (homogeneous)
  - Referencing
    - ⇔ Record (by identifiers), Array (by indices)

(1) The Structure of Records

- In COBOL,

nested structure

in data division

level number

```
01    EMPLOYEE-RECORD.
   02    EMPLOYEE-NAME.
         05  FIRST            PICTURE IS X(20).
         05  MIDDLE           PICTURE IS X(10).
         05  LAST             PICTURE IS X(20).
   02    HOURLY-RATE          PICTURE IS 99V99.
```

- In Ada (Modular-2)

```
EMPLOYEE_RECORD :
    record
        EMPLOYEE_NAME :
            record
                FIRST   : STRING (1..20) ;
                MIDDLE : STRING (1..10) ;
                LAST : STRING (1..20) ;
            end record ;
        HOURLY_RATE : FLOAT ;
    end record ;
```

- In FORTRAN and C
  – Nested된 record는 먼저 선언하고, nested된 경우 그 이름을 쓴다.

```
struct aa {
   int a ;
   char b ;
}

struct bb {
   int c ;
   struct aa d ;
} ff ;
```


(2) References to Record Fields

- Syntax
  – In COBOL (using “OF”),

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

  – In most other languages (using “.”),

```
EMPLOYEE_RECORD.EMPLOYEE_NAME.MIDDLE
```

- Referencing Method
  – *Fully qualified reference* : all intermediate record names, from the largest enclosing record to the specific field, are named in the references
  – *Elliptical reference* : the field is named, but any or all of the enclosing record names can be omitted, if it is unambiguous in the referencing environment
- In Pascal,

```
employee.name := 'Bob' ;
employee.age := 42 ;
employee.sex := 'M' ;
```

$\approx$

```
with employee do
    begin
        name := 'Bob' ;
        age := 42 ;
        sex := 'M' ;
    end ;
```

(3) Operations on Records

- In Pascal and Modular-2, record can be assigned
- Ada allows record assignment and comparison for equality and inequality
- In COBOL,

```
01 INPUT-RECORD.
    02 NAME.
          03 LAST     PIC IS X(20).
          03 MIDDLE   PIC IS X(15).
    02 HOURS-WORKED   PIC IS 99.
     .....
```

```
01 OUTPUT-RECORD.
    02 NAME.
          03 LAST     PIC  IS X(20).
          03 MIDDLE   PIC IS X(15).
   02 NET-PAY    PIC    IS  999v99.
     .....
```

```
MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD
```

copies the fields of specified source record to the destination record only if the destination record has the field with the same name

(4) Implementation

- The fields of records are stored in adjacent memory location
  - *Access method* : the offset address, relative to the beginning of the record, is associated with each field (word alignment)
- Compile time descriptor for a record

Record
name
type
offset
....

field 1

```
Struct aa {
    char c ;
    int i ;
} a;

a.i = 3 ;
```

1000번지

Address ?

- Evaluation and Comparison to Arrays
  - Records are used when collection of data values is heterogeneous
  - Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)

6.8 Tuple Types

- A tuple
  - is a data type that is similar to a record, except that the elements are not named
  - Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - ⇔ Closely related to its lists, but immutable
    - ⇔ Create with a tuple literal

```
myTuple = (3, 5.8, 'apple')
```

      - Referenced with subscripts (begin at 1)
      - Catenation with + and deleted with `del`
  - ML

```
val myTuple = (3, 5.8, 'apple');
```

    - ⇔ access as follows: `#1(myTuple)` is the first element
    - ⇔ a new tuple type can be defined

```
type intReal = int * real;
```


6.9 List Types

- Lists in LISP and Scheme are delimited by parentheses and use no commas

```
(A B C D) and (A (B C) D)
```

- Data and code have the same form
  - as data, `(A B C)` is literally what it is
  - as code, `(A B C)` is the function `A` applied to the parameters `B` and `C`
  - The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

```
'(A B C) is data
```

- List Operations in Scheme (LISP)
  - CAR returns the first element of its list parameter
    - ⇔ `(CAR '(A B C))` returns `A`
  - CDR returns the remainder of its list parameter after the first element has been removed
    - ⇔ `(CDR '(A B C))` returns `(B C)`
  - CONS puts its first parameter into its second parameter, a list, to make a new list
    - ⇔ `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` returns a new list of its parameters
    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

- **Examples**
  - `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
DEFINE (member atm a_list)
        (COND
                ((NULL? a_list) #F)
                ((EQ? atm (CAR lis)) #T)
                ((ELSE (member atm (CDR a_list)))
        )
)
```

  - `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append list1 list2)
        (COND
                ((NULL? list1) list2)
                (ELSE (CONS (CAR list1)
                           (append (CDR list1) list2)))
        )
)
```


- **Python Lists**
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common LISP, ML, and F#, Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

  - List elements are referenced with subscripting, with indices beginning at zero

```
x = myList[1]
```

Sets `x` to `5.8`

  - List elements can be deleted with `del`

```
del myList[1]
```

  - List Comprehensions – derived from set notation

```
[x * x for x in range(6) if x % 3 == 0]
```

`range(6)` creates `[0, 1, 2, 3, 4, 5, 6]`

Constructed list: `[0, 9, 36]`

List vs. Array ??

6.10 UNION Types

– a union is a type that is allowed to store different type values at different times during program execution

⇔ Example : a table of constant for a compiler

Alias *vs.* Union	unsafe flexible

(1) Design Issues : type checking of union types

- Should type checking be required ? (must be dynamic)
- Should union be embedded in records ?
- In Fortran,
  - `EQUIVALENCE` primitive
  - No type checking

```
INTEGER X
REAL Y
EQUIVALENCE (X,Y)
```

both X and Y are to cohabit the same storage address (alias)

conformity union

```
union (int, real)  ir1 ;
int count ;
real sum ;
...
case ir1 in
     (int intval) : count := intval ;
     (real realval) : sum := realval ;
esac
```

- ALGOL 68 Union Types
  - *Discriminated union* :

    ⇔ a union with which is associated an additional value called a *tag*, or *discriminant*, that identifies the current type value stored in the union

    ⇒ can be used for type checking in run-time
  - *Conformity clause* : the correct choice is made by testing a type tag maintained by the run-time system for the variable (free union ➔ C, C++)

(2) Pascal Union Types → *integrating discriminant unions with a record structure*

- *Variant record* : record structure type with a discriminated union
  - The discriminant is a user-accessible variable in the record that stores the current type value in the variant

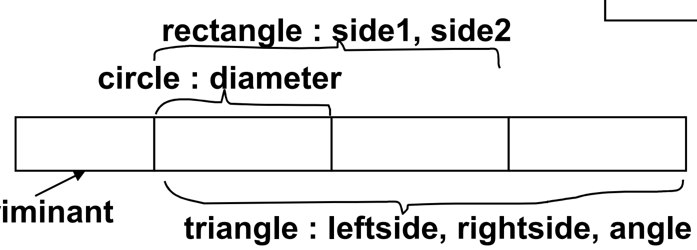
```
type shape = (circle, triangle, rectangle) ;
  object =
    record
       case form : shape of
         circle :    (diameter:real) ;
         triangle : (leftside:integer, rightside:integer; angle:real);
         rectangle : (side1 : integer ;side2 : integer)
           end ;
var   thing : object
......................
case thing.form of
     circle :       aa := thing.diameter;
     triangle :     aa := thing.leftside ;
     rectangle :    aa := thing.side1 ;
end
```

discriminant

• C , C++ : Free Union

no tag, no type checking

Storage Allocation



- Problems
  - Inconsistency between tag and value
  - Free union
  - Hard to check the type

```
thing.diameter := 2.73 ;
side := thing.leftside ;
```

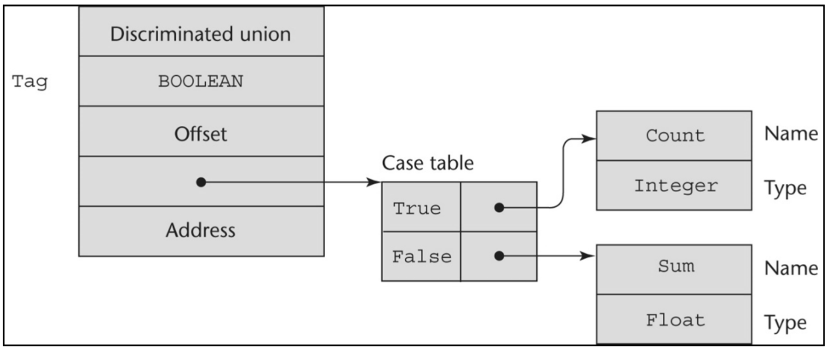
(3) Implementation

- **Discriminated unions are implemented by simply using the same address for every possible variant**
  - **Sufficient storage for the largest variant is allocated**

Ada

```
type NODE (TAG : BOOLEAN) is
   record
      case TAG is
         when TRUE => COUNT : Integer
         when false => SUM : char
```

→ ***storage allocation ?***



- **Java and C# do not support unions**
  - **Reflective of growing concerns for safety in programming language**

6.11 Set Types

- **A *set* type is one whose variables can store unordered collections of distinct values from some ordinal type called its *base type***
- ***Design Issues***
  - **What should be the maximum number of elements in a set base type ?**

(1) Sets in Pascal

- **In Pascal,**
  - **The maximum size of Pascal base sets is *implementation dependent* (usually less than 100)**

    ⇔ Sets and their operations are most efficiently implemented by representing set variables as bit strings that fit into a single machine word
  - **Pascal set operations**

    := assignment of compatible set types
    + set union
    *** set intersection**
    - set difference
    = set equality

```
type colors = (red, blue, green, yellow, orange, white) ;
     colorset = set of colors ;
var set1, set2, set3 : colorset ;

set1 := [red, blue, yellow, white] ;
set2 := [blue, yellow] ;
set3 = set1 + set2 ;
set3 = set1*set2 ;
if (red in set1) then .....
```

  - **Set vs. Array**

    ⇔ Set operations are more efficient than equivalent array operations

(2) Implementation

- Sets are usually stored as bit strings in memory
  – Example : if a set has ordinal base type ['a'..'o'], then this set type can use the fit 15 bit of a machine word, and each set bit(1) representing a present element, and each clear bit (0) representing an absent element
  – set union : a logical OR
  – member check: a logical AND

```
kk = ['a','c','h','o']   10100001000000
if ('b' in kk) ……
                     AND 01000000000000
```


6.12 Pointer and Reference Types

- A *pointer* type is one in which the variables have a range of values that consists of memory addresses and a special value, *nil*
  – Used for indirect addressing and dynamic storage management

(1) Design Issues

- What are the scope and lifetime of a pointer variable ?
- What is the lifetime of a dynamic variable ?
- Are pointer restricted as to the type of object to which they can point ?

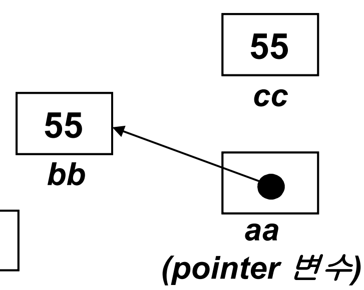
(2) Pointer Operations

- Two fundamental pointer operations
  – Assignment : Set a pointer variable to the address of some object

```
int  *aa, bb, cc ;
aa = &bb ;
```

  – Dereferencing : yields the value stored at the location represented by the pointer's value

```
cc = *aa ;
```



(3) Pointer and Pointer Problems in PL/I

- **Type Checking**
  - **The type of object to which a pointer can point is called it domain type. A PL/I pointer is not restricted to a single domain type**
- **Dangling Pointer**
  - **A *dangling pointer* is a pointer that contains the address of dynamic variable that has been deallocated**
- **Lost Objects**
  - **A *lost object (garbage)* is an allocated dynamic object that is no longer accessible to the user program but may still contain useful data**

```
int *i ;
…
sub1() {
   int j;
   j = 5;
   i = &j
}

*i = ??
```

dangling pointer

```
char *c ;
c = malloc(…) ;
…
c = malloc(…);
```

lost object pointer

(4) Pointers in Pascal

- **In Pascal, pointers are used only to access dynamically allocated anonymous variables**
  - **allocation : `new`**
  - **deallocation : `dispose`**
- **Implementation of the function `dispose`**
  - **Simply ignore dispose**
  - **Do not include dispose in the language**
  - **Allow the dangling pointer**
  - **Find and set all pointers pointing to the dynamic variable being destroyed to nil -> hard to implement**
  - **Implement dynamic storage allocating using stack (mark and release the stack (used heap))**

***"The introduction of pointer to high-level languages has been a step backward from which we may never recover"* [Hoare 1973]**

goto : control
pointer : data

(5) Pointers in C

- **Pointers can be used much like addresses are used in assembly language**
- **Operators**
  - **'&' : used for producing the address of variable**
  - **'*' : used for dereferencing**
  - **Pointer arithmetic is possible**
    - **⇔"ptr + index" : Instead of simply adding the value of index to ptr, the value of index is first scaled by the size of the object (in memory units) to which ptr is pointing**

```
char *c ;
int *i;

*(c + 1)
*(i + 1)
```

(6) Reference Types

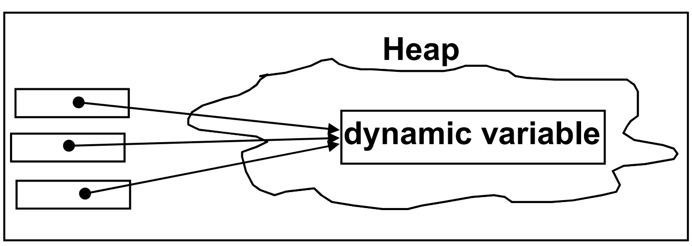
- Pointer *vs.* Reference
  - pointer : refers to an address in memory
  - reference : refers to an object or value in memory
- Reference type variable in C++
  - a constant pointer that is always implicitly dereferenced
  - a C++ reference type variable is a constant
    - ⇔ it must be initialized with the address of some variable in its definition
    - ⇔ after initialization, a reference type variable can never be set to reference any other variable
    - ⇔ used to two-way communication in function call

```
int result = 0 ;
int &ref_result = result ;
…
ref_result = 100 ; /* result and ref_result are alias */
```

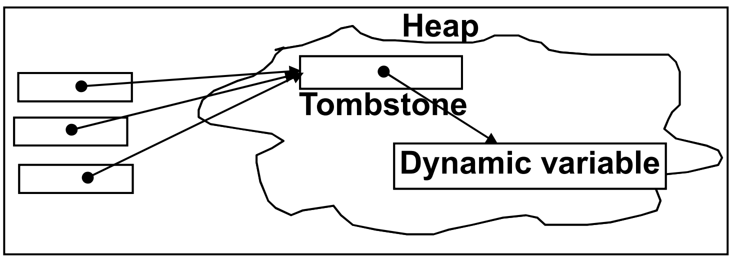
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses

(7) Implementation

- Representations of Pointer
  - In most large computers, *pointers* are single values stored in either two- or four-byte memory cells (word size)
- Solutions to the ***Dangling Pointer Problem***
  1) ***Tombstone Approach***
     - ⇔ the idea is to have all dynamic variable include a special cell, called a ***tombstone***, that is itself a pointer to the dynamic variable
     - ⇔ When a dynamic variable is deallocated, the tombstone remains but is set to *nil*, indicating that ***the dynamic variable no longer exists***
     - ⇔ It is ***costly in both time and space***
     - ⇔ used extensively by Macintosh system



Without tombstone



With tombstone

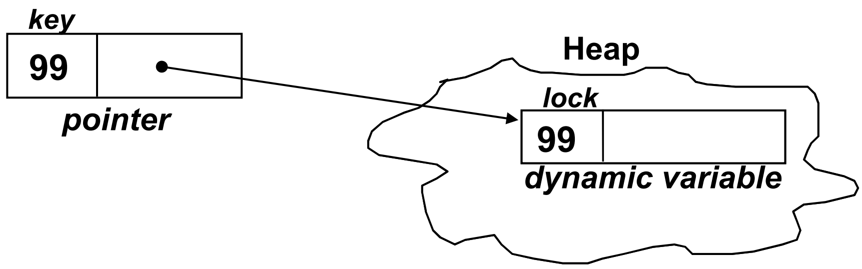
2) Locks-and-key Approach

⇔ the pointer values are represented as ordered pair (key, address) while key is an integer value

⇔ dynamic variables are represented as the storage for the variable plus a header cell that stores an integer lock value, and any copies of the pointer value to other pointer must copy the key value

⇔ every access to the dereferenced pointer compares the key value of the pointer to the lock value in the dynamic variable

⇔ deallocation of variable clears the lock value



- **Heap Management**
  - Assumption : a fixed-size allocation heap (LISP)
  - all available cells can be linked together, forming a list of available space
  - If deallocation is implicit, then, when should deallocation be performed ?

⇔ Reference Counter approach

⇒ incremental reclamation

⇔ Garbage Collection approach

⇒ batch reclamation

1) Reference Counter Approach

⇔ maintaining a counter in every cell, which stores the number of pointers that are currently pointing at cell

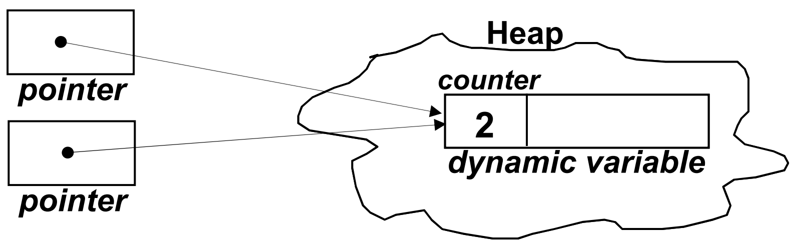
⇔ reclamation is incremental and is done when reference counter reaches zero

⇔ Problems

⇒ space requirement for the counter

⇒ execution time to maintain the counter

⇒ circular reference



2) Garbage Collection of fixed size cell

⇔ the run-time system allocates storage cells as requested and disconnects pointers from cell as necessary, without regard for storage reclamation, until it has allocated all available cells

⇔ at this point, a garbage collection process is begun to gather all the garbage left floating around the heap

⇔to facilitate the garbage collection process, every heap cell has an extra indicator bit or field that is used by the collection algorithm

⇔ Simple algorithm

1) All cells in the heap have their indicators set to indicate they are garbage

2) Every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage

3) All cells in the heap that have not been specifically marked as not being garbage are returned to the list of available space

⇔ Problems in Garbage Collection Approach

⇒ <u>When you need it most, it works the worst</u>

⇒ Space for mark bit, and time to trace

3) Garbage Collection of variable size cells

⇔ Additional difficulties

1) Initial setting of indicators of all cells in the heap to indicate that they are garbage is difficult

2) The marking process is nontrivial, because of the cell without pointer

3) Fragmentation of available spaces

- 주소 계산 예제

다음과 같은 C-like한 언어 프로그램이 Intel 80386 (32bit 처리기)을 사용하는 컴퓨터에서 수행된다고 가정하고 다음의 물음에 답하라. (단, word-alignment를 하며, set 변수를 저장할 시 꼭 필요한 byte 수만을 할당하며, array는 column-major형태로 저장된다고 가정한다. 또한 union은 discriminated union으로 가정한다.)

(1) lala 변수를 저장시키는데 필요한 메모리 양은 몇 byte인가 ? (10점)

(2) test[10][10]의 메모리 상에서의 시작 주소가 1000번지라고 가정하였을 때,
“&(test[4][6].ColClass.j) + 2”이 지정하는 주소는 어디인가 ? (10점)

```
type colors = (red,blue);
type colorset = set of colors ;
type index = 1..200;

struct {
  index i ;
  colorset mycolor ;
  case ColClass : colors of {
      red : {char c; float j;}
      blue : {int i;}
  }
} test[index][index], lala ;
```

1000 + ((4*200)+6)*20 + 16 + 8

- **Homework**
  - Analyze the assembly code of the program with 3D-array, record, enumeration type, union, floating point, and pointer in your favorite language.
    - ⇔ storage allocation
    - ⇔ referencing mechanism

test.c

```
#include <stdio.h>

int a[10][10][10] ;
struct AA {
    int          a ;
    char         b ;
    float        c;
} aa ;

enum days {Sun, Mon, Tue, Wed,
           Thu, Fri, Sat} d1;

union BB {
   char a ;
   int b;
} bb ;

int *cc, k ;

main()  {
  k = a[4][5][7] ;
  d1 = Sun ;
  bb.b = k ;
  cc = &bb.b ;
}
```

test.s

```
.file   "test.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl    %ebp
        movl     %esp,%ebp
        movl     a+1828,%eax
        movl     %eax,k
        movl     $0,d1
        movl     k,%eax
        movl     %eax,bb
        movl     $bb,cc
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .comm    a,4000,32
        .comm    aa,12,4
        .comm    d1,4,4
        .comm    bb,4,4
        .comm    cc,4,4
        .comm    k,4,4
        .ident   "GCC: (GNU) 2.95.1(release)"
```

`% cc -S test.c`