Chapter 6

Structured Data Types

Topics

- Vectors
- Arrays
- Slices
- Associative Arrays
- Records
- Unions
- Lists
- Sets

Structured Data Types

- Virtually all languages have included some mechanisms for creating complex data objects:
  - Formed using elementary data objects.
  - Arrays, lists, and sets are ways to create homogeneous collection of objects.
  - Records are a mechanism for creating nonhomogeneous collections of data objects.

Vectors and Arrays

- Vectors and arrays are the most common types of data structures in programming languages.
- A *vector* is a data structure composed of a fixed number of components of the same type organized as a simple linear sequence.
- A component of a vector is selected by giving its *subscript,* an integer (or enumeration value) indicating the position of the component in the sequence.
- A vector is also called a *one-dimensional array* or *linear array.*

Vectors

- The attributes of a vector are:
  1. *Number of components*: usually indicated implicitly by giving a sequence of subscript ranges, one for each dimension.
  2. *Data type of each component*, which is a single data type, because the components are all of the same type.
  3. *Subscript to be used to select each component*: usually given as a range of integers, with the first integer designating the first component, and so on.

Vectors: subscripts

- Subscripts may be either a range of values as -5...5 or an upper bound with an implied lower bound, as A(10).
- Examples:
  - In Pascal, V: **array** [-5 .. 5] **of** real;

    Defines a vector of 11 components, each a real number, where the components are selected by the subscripts, -5, -4, ... 5.
  - In C, float a[10];

    Defines a vector of 10 components with subscripts ranging from 0 to 9.

Vectors: subscripts

- Subscript ranges need not begin at 1.
- Subscript ranges need not even be a subrange of integers; it may be any enumeration (or a subsequence of an enumeration)
- Example:
  - In Pascal, **type** class = (Fresh, Soph, Junior, Senior);
    var ClassAverage: **array** [class] **of** real;

Vectors: operations

- *Subscripting*: the operation that selects a component from a vector.
  - It is usually written as the vector name followed by the subscript of the component to be selected.
    - V[2] or ClassAverage[Soph]
  - It may be a computed value (an expression that computes the subscript)
    - V[I + 2]

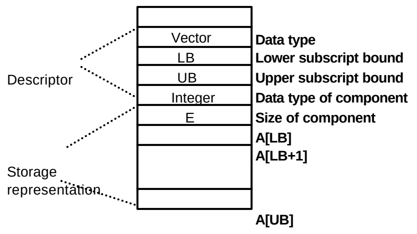
Vectors: other operations

- Operations to create and destroy vectors.
- Assignment to components of a vector.
- Operations that perform arithmetic operations on pairs of vectors of the same size (i.e. addition of two vectors).
- Insertions and deletions of components are not allowed
  - Only the value of a component may be modified.

Vectors: implementation

- Storage and accessing of individual components are straightforward:
  - Homogeneity of components
    - The size and structure of each component is the same.
  - Fixed size
    - The number and position of each component of a vector are invariant through its lifetime.
- A sequential storage is appropriate.

Vectors: implementation



Vectors: access function

- An access function is used to map array subscripts to addresses.
- can be addressed by skipping I-1 components.
  - If E is the size of each component, then skip (I-1) x E memory locations.
  - If LB is the lower bound on the subscript range, then the number of such components to skip is I-LB or (I-LB) x E memory locations.

Vectors: access function

- If the first element of the vector begins at location α, the access function is:

  $$address(A[I]) = \alpha + (I - LB) \times E$$

  which can be rewritten as:

  $$address(A[I]) = (\alpha - LB \times E) + (I \times E)$$

- Once the storage for the vector is allocated, $(\alpha - LB \times E)$ is a constant (K) and the accessing formula reduces to

  $$address(A[I]) = K + I \times E$$

- Example: access function of a C vector

  address(A[I]) = address(array[0]) + i*element_size

Multidimensional Arrays

- An array is a homogeneous collection of data elements in which an element is identified by its position in the collection, relative to the first element
- *Indexing* is a mapping from indices to elements

  map(array_name, index_value_list) $\rightarrow$ an element
- Indexes are also known as *subscripts*.
- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most other languages use brackets

Arrays: subscript types

- What type(s) are allowed for defining array subscripts?
  - FORTRAN, C, C++, and Java allow integer subscripts only.
  - Pascal allows any ordinal type
    - int, boolean, char, enum
  - Ada allows integer or enumeration types
    - Including boolean and char

Arrays: subscript issues

- In some languages the lower bound of the subscript range is implicit
  - C, C++, Java—fixed at 0
  - FORTRAN—fixed at 1
  - VB (0 by default, could be configured to 1)
- Other languages require programmer to specify the subscript range.

Arrays: 4 categories

- There are 4 categories of arrays based on subscript range bindings and storage binding:
  - Static
  - Fixed stack-dynamic
  - Stack dynamic
  - Heap-dynamic

Static Arrays

- Static arrays are those in which
  - Range of subscripts is statically bound (at compile time).
  - Storage bindings are static (initial program load time)
- Examples:
  - FORTRAN77, global arrays in C, static arrays (C/C++), some arrays in Ada.
- Advantage:
  - Execution efficiency since no dynamically allocation/deallocation is required
- Disadvantages:
  - Size must be known at compile time.
  - Bindings are fixed for entire program.

Fixed stack dynamic Arrays

- Fixed stack-dynamic arrays are those in which
  - Subscript ranges are statically bound.
  - Allocation is done at declaration elaboration time (on the stack).
- Examples:
  - Pascal locals, most Java locals, and C locals that are not static.
- Advantage is space efficiency
  - Storage is allocated only while block in which array is declared is active.
  - Using stack memory means the space can be reused when array lifetime ends.
- Disadvantage
  - Size must be known at compile time.

Stack dynamic Arrays

- A stack-dynamic array is one in which
  - Subscript ranges are dynamically bound
  - Storage allocation is done at runtime
  - Both remain fixed during the lifetime of the variable
- Advantage: flexibility - size need not be known until the array is about to be used
- Disadvantge: once created, array size is fixed.
- Example:
  - Ada arrays can be stack dynamic:

```
Get(List_Len);
Declare
  List : array (1..List_Len) of Integer;
Begin
…
End;
```

Heap dynamic Arrays

- Storage is allocated on the heap
- A heap-dynamic array is one in which
  - Subscript range binding is dynamic
  - Storage allocation is dynamic
- Examples:
  - 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: Array Bindings

- Binding times for Array

	Subscript range	*Storage*
Static	Compile time	Compile time
Fixed stack dynamic	Compile time	Declaration elaboration time
Stack dynamic	Runtime but fixed thereafter	Runtime but fixed thereafter
Dynamic	Runtime	Runtime

Arrays: attributes

- Number of scripts
  - FORTRAN I allowed up to three
  - FORTRAN 77 allows up to seven
  - Others languages have no limits.
  - Other languages allow just one, but elements themselves can be arrays.
- Array Initialization
  - Usually just a list of values that are put in the array in the order in which the array elements are stored in memory

Arrays: initialization

- Examples of array initialization:
  1. FORTRAN - uses the DATA statement, or put the values in `/ ... /` on the declaration

     ```
     Integer, Dimension (4) :: stuff = (/2, 4, 6, 8/)
     ```
  2. Java, C and C++ - put the values in braces; can let the compiler count them

     e.g.

     ```
     int stuff [] = {2, 4, 6, 8};
     ```
  3. For strings (which are treated as arrays in C and C++), an alternate form of initialization is provided.

     ```
     char* names[] = {"Bob", "Mary", "Joe"};
     ```

Arrays: initialization

3. Ada provides two mechanisms
   - List in the order in which they are stored.
   - Positions for the values can be specified.

```
stuff : array (1..4) of Integer := (2,4,6,8);
SCORE : array (1..14, 1..2) of Integer := (1 => (24,
10), 2 => (10, 7), 3 =>(12, 30), others => (0, 0));
```

4. Pascal does not allow array initialization

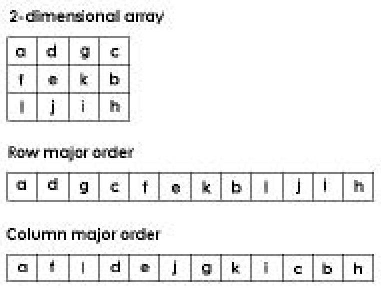
Arrays: implementation

- A matrix is implemented by considering it as a vector of vectors; a three-dimensional arrays is a vector whose elements are vectors, and so on.
  - All subvectors must have the same number of elements of the same type.
- Matrix:
  - Column of rows vs. row of columns

Arrays: implementation

- *Row-major order* (column of rows)
  - The array is first divided into a vector of subvectors for each element in the range of the first subscript, then each of these subvectors is subdivided into subvectors for each element in the range of the second subscript, and so on.
- *Column-major order (single row of columns)*

Arrays: Row- vs. Column-major order


2-dimensional array

a	d	g	c
f	e	k	b
i	j	i	h

Row major order

a	d	g	c	f	e	k	b	i	j	i	h

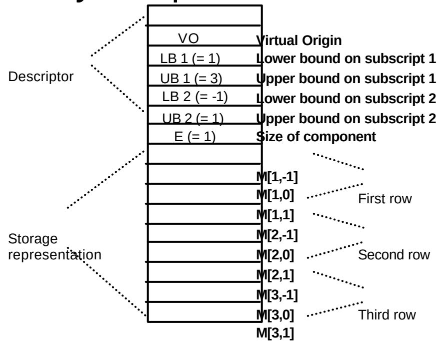
Column major order

a	f	i	d	e	j	g	k	i	c	b	h

Arrays: storage representation

- Storage representation follows directly from that for a vector.
  - For a matrix, the data objects in the first row (assuming row -major order) followed by the data objects in the second row, and so on.
  - Result: a single sequential block of memory containing all the components of the array in sequence.
    - The descriptor is the same as that for the vector, except that an upper and lower bound for the subscript range of each dimension are needed.

Arrays: implementation



Arrays: accessing function

- The accessing function is similar to that for vectors:
  - Determine the number of rows to skip over $(I-LB_1)$
  - Multiply by the length of a row to get the location of the start of the I^{th} row
  - Find the location of the J^{th} component in that row, as for a vector

Arrays: accessing function

- If A is a matrix with M rows and N columns, the location of element A[I,J] is given by:
  - A is stored in row-major order

    $$\text{location}(A[I,J]) = \alpha + (I - LB_1) \times S + (J - LB_2) \times E$$

    where S = length of a row = $(UB_2 - LB_2 + 1) \times E$
  - A is stored in column-major order

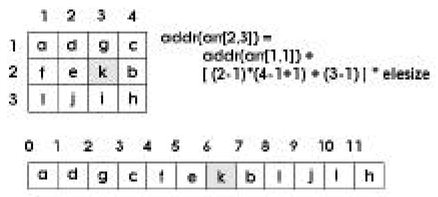
    $$\text{location}(A[I,J]) = \alpha + (J - LB_2) \times S + (I - LB_1) \times E$$

    where S = length of a row = $(UB_1 - LB_1 + 1) \times E$

where α = base address
LB_1 = lower bound on first subscript
LB_2, UB_2 = lower and upper bounds on the second subscript

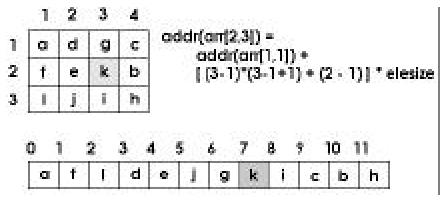
Arrays: Row-major access function

$$\text{location}(A[I,J]) = \alpha + (I - LB_1) \times S + (J - LB_2) \times E$$
$$S = (UB_2 - LB_2 + 1) \times E$$



Arrays: Column-major access function

$$\text{location}(A[I,J]) = \alpha + (J - LB_2) \times S + (I - LB_1) \times E$$
$$S = \text{length of a row} = (UB_1 - LB_1 + 1) \times E$$



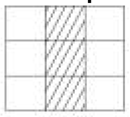
Slices

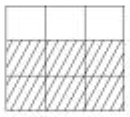
- A slice is some substructure of an array
  - Nothing more than a referencing mechanism
  - A way of designating a part of the array
- Slices are only useful in languages for operations that can be done on a whole array.

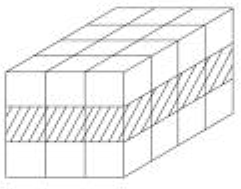
1. In FORTRAN 90

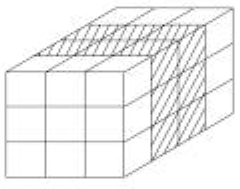
   INTEGER MAT (1:4, 1:4)
   MAT(1:4, 1) - the first column
   MAT(2, 1:4) - the second row

Example Slices in FORTRAN 90


MAT (1:3, 2)


MAT (2:3, 1:3)


CUBE (2, 1:3, 1:4)


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

Slices: examples

2. Ada - single-dimensioned arrays only

   `LIST(4..10)`

3. Java has something like slices for multi-dimensioned arrays

   `int [][]array = array[1]` - gets the second row

- PL/I was one of the earliest languages to implement slices.

Associative Arrays

- An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys
- The keys are stored in the structure
- Thus, element is a (key, value) pair
- Design Issues:
  1. What is the form of references to elements?
  2. Is the size static or dynamic?

Associative Arrays

- Structure and Operations in Perl
  - Names begin with %

```
%hi_temps =    ("Monday" => 77,
                "Tuesday" => 79,
                "Wednesday" => 83);
```

  - Alternative notation

```
%hi_temps =    ("Monday", 77,
                "Tuesday", 79,
                "Wednesday", 83);
```

Associative Arrays

- Structure and Operations in Perl
  - Subscripting is done using braces and keys

```
$hi_temps{"Wednesday"};
#returns the value 83
```

  - A new elements is added by

```
$hi_temps{"Thursday"} = 91;
```

  - Elements can be removed with **delete**

```
delete $hi_temps{"Thursday"};
```

Records

- A data structure composed of a fixed number of components of different types.
- Vectors vs. Records
  1. The components of records may be *heterogeneous*, of mixed data types, rather than homogeneous.
  2. The components of records are named with *symbolic names* (identifiers) rather than indexed with subscripts.

Records: attributes

- Records have 3 main attributes:
  1. The number of components
  2. The data type of each component
  3. The selector used to name each component

  - The components of a records are often called *fields*, and the component names then are *field names*.
  - Records are sometimes called *structures* (as in C).

Records: definition syntax

1. COBOL uses level numbers to show nested records

```
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
```

Records: definition syntax

2. Other languages use recursive definitions

```
  type Employee_Name_Type is record
          First   :  String(1..20);
          Middle: String (1..10);
          Last :     String (1..20);
  end record;
  type Employee_Record_Type is record
          Employee_Name: Employee_Name_Type;
          Hourly_Rate: Float;
  end record;
Employee_record: Employee_Record_Type;
```

Record Field References

1. COBOL

field_name OF record_name_1 OF ... OF record_name_n

Example:

MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

2. Others (dot notation)

record_name_1.record_name_2. ... record_name_n.field_name

Example:

Employee_Record.Employee_Name.Middle

Records

- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous (Cobol only)
- Pascal and Modula-2 provide a `with` clause to abbreviate references

Records: operations

- Assignment
  - Pascal, Ada, and C++ allow it if the types are identical.
- Initialization
  - Allowed in Ada, using an aggregate.
- Comparison
  - In Ada, = and /=; one operand can be an aggregate
- Move Corresponding
  - In COBOL: it moves all fields in the source record to fields with the same names in the destination record.

Records: initialization

- Define & initialize with list of variables

```
struct student s1 =
  {"Ted","Tanaka",
  22, 2.22};
```

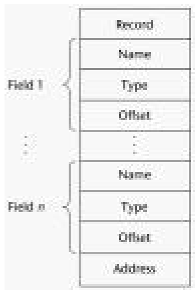
- Define & initialize using the dot operator (structure member operator)

```
struct student s2;
strcpy(s.first,
     "Sally");
s.last="Suzuki";
s.age = 33;
s.gpa = 3.33;
```

Records: implementation

- The storage representation for a record consists of a single sequential block of memory in which the components are stored in sequence.
- Individual components may need descriptors to indicate their data type and other attributes.
  - No runtime descriptor for the record is required.

Records: descriptors



Unions

- A union is a type whose variables are allowed to store different type values at different times during execution
- Design Issues for unions:
  1. What kind of type checking, if any, must be done?
  2. Should unions be integrated with records?

Unions: examples

1. FORTRAN - with **EQUIVALENCE**

   EQUIVALENCE (A, B, C, D), (X(1), Y(1))

   - *Free Unions:*
     - No tag variable is required.
     - No type checking
     - C/C++ have free unions
2. Pascal: variant records
   - Contain one or more components that are common to all variants.
   - Each variant has several other components with names and data types that are unique to each variant.

Unions: examples

```
type PayType = (Salaried, Hourly);
var Employee: record
     ID: integer;
     Dept: array [1..3] of char;
     Age: integer;
     case PayClass: PayType of
            Salaried: (MontlyRate: real;
                       StartDate: integer);
            Hourly: (HourRate: real;
                     Reg: integer;
                     Overtime: integer)
     end
```

- The component *PayClass* is called the *tag* (Pascal) or *discriminant* (Ada) because it serves to indicate which variant of the record exists at a given point during program execution.

Unions: type checking issues

- System must check value of flag before each variable access

```
Employee. PayClass := Salaried;
Employee. MontlyRate := 1973.30;
...
print(Employee.Overtime);          -- error
```

- Still not good enough!

```
Employee. PayClass := Salaried;
Employee. MontlyRate := 1973.30;
Employee. StartDate := 626;

Employee. PayClass := Hourly;
print(Employee.Overtime);          -- this should be an error
```

Unions: selection operation

- Selection operation: same as that for an ordinary record.
  - For ordinary records: each component exists throughout the lifetime of the record.
  - For variant records/unions: the component may exist at one point during execution (when the tag component has a particular value), may later cease of exist (when the value of the tag changes to indicate a different variant), and later may reappear (if the tag changes back to its original value).

Ada Union Types

- Similar to Pascal, except
  - No free union
    - Tag must be specified with union declaration
  - When tag is changed, all appropriate fileds must be set too.

```
Employee.PayClass := Hourly;
Employee.HourRate := 8.75;
Employee.Reg := 8;
Employee.Overtime := 2;
```

  - Ada union types are safe
    - Ada systems required to check the tag of all references to variants.

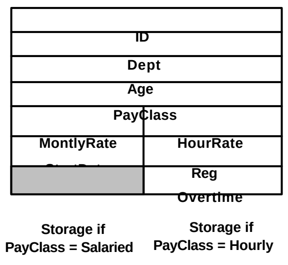
Unions: type checking

- Problem with Pascal's design
  - Type checking is ineffective.
  - User can create inconsistent unions (because the tag can be individually assigned)
  - Also, the tag is optional (free union).
- Ada discriminant union
  - Tag must be present
  - All assignments to the union must include the tag value –tag cannot be assigned by itself.
  - It is impossible for the user to create an inconsistent union.

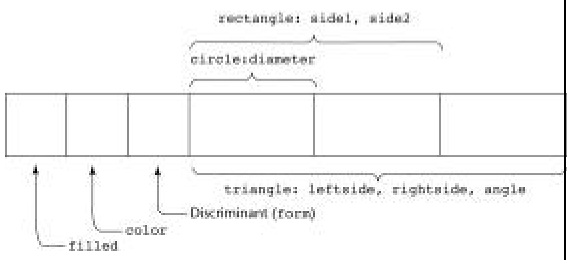
Unions: implementation

- During translation, the amount of storage required for the components of each variant is determined
  - Storage is allocated in the record for the *largest* possible variant.
  - Each variant describes a different layout for the block in terms of number and types of components.
- During execution, no special descriptor is needed for a variant record because the tag component is considered just another component of the record.

Unions: storage representation



Unions: storage representation



Union: evaluation

- Useful
- Potentially unsafe in most languages
- Ada, Algol 68 provide safe versions

Pointers: problems

1. Dangling pointers (dangerous)

- A pointer points to a heap-dynamic variable that has been deallocated
- Creating one (with explicit deallocation):
  a. Allocate a heap-dynamic variable and set a pointer to point at it
  b. Set a second pointer to the value of the first pointer
  c. Deallocate the heap-dynamic variable, using the first pointer

Pointers: problems

- Dangling pointers

```
int *p, *q;
p = (int*)malloc(sizeof(int* 5);
q = p;
free (p);
```

2. Lost Heap-Dynamic Variables (wasteful)

- A heap-dynamic variable that is no longer referenced by any program pointer
- Creating one:
  a. Pointer p1 is set to point to a newly created heap-dynamic variable

Pointers: problems

b. p1 is later set to point to another newly created heap-dynamic variable

- The process of losing heap-dynamic variables is called memory leakage
- Lost heap-dynamic variables (garbage)

```
p = (int*)malloc(5*sizeof(int));
p = new int(20);
```

The process of losing heap-dynamic variables is called **memory leakage** . (cannot free the first chunk of memory)

Pointers: examples

- C and C++ pointers
  - Used for dynamic storage management and addressing
  - Explicit dereferencing (*) and address-of operator (&)
  - Can do pointer arithmetic

```
float arr[100];
float *p = arr;
*(p+5) ° arr[5] ° p[5]
*(p+i) ° arr[i] ° p[i]
```

  - **void*** can point to any data type but cannot be dereferenced

Pointers: examples

- C++ reference types
  - Constant pointers that are implicitly dereferenced:

```
float x = 1.0;
float &y = x;
y = 2.2; → sets x to 2.2
```

  - Used for reference parameters:
    - Advantages of both pass-by-reference and pass-by-value

Pointers: examples

- Java - Only references (no pointers)
  - No pointer arithmetic
  - Can only point at objects (which are all on the heap)
  - No explicit deallocator (garbage collection is used)
  - Means there can be no dangling references
  - Dereferencing is always implicit

Lists

- A data structure composed of an ordered sequence of data structures.
- List are similar to vectors in that they consist of an ordered sequence of objects.
- Lists vs. Vectors
  1. Lists are rarely of fixed length. Lists are often used to represent arbitrary data structures, and typically lists grow and shrink during program execution.
  2. Lists are rarely homogeneous. The data type of each member of a list may differ from its neighbour.
  3. Languages that use lists typically declares such data implicitly without explicit attributes for list members.

Variations on Lists

- Stacks and queues
  - A *stack* is a list in which component selection, insertion, and deletion are restricted to one end.
  - A *queue* is a list in which component selection and deletion are restricted to one end and insertion is restricted to the other end.
  - Both sequential and linked storage representations for stacks and queues are common.

Variations on Lists

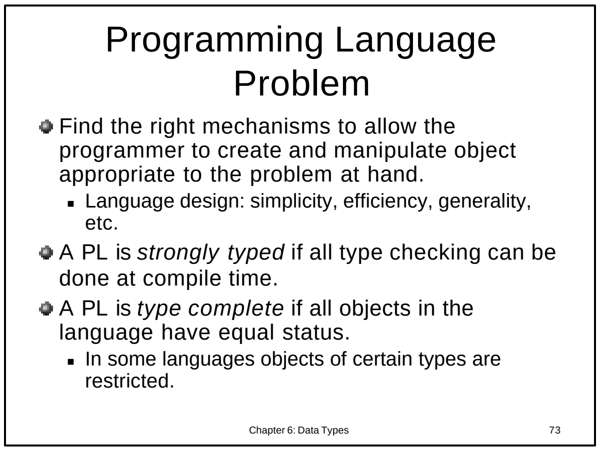
- Trees
  - A list in which the components may be lists as well as elementary data objects, provided that each list is only a component of at most one other list.
- Directed graphs
  - A data structure in which the components may be linked together using arbitrary linkage patterns (rather than just linear sequences of components).

Variations on Lists

- Property lists
  - A record with a varying number of components, if the number of components may vary without restriction
  - The component names (property names) and their values (property values) must be stored.
  - A common representation is an ordinary linked list with the property names and their values alternating in a single long sequence.

Sets

- A set is a data object containing an unordered collection of distinct values.
- Basic operations on sets:
  1. *Membership*.
  2. *Insertion and deletion of single values.*
  3. *Union of sets*
  4. *Intersection of sets*
  5. *Difference of sets*



Programming Language Problem

- Find the right mechanisms to allow the programmer to create and manipulate object appropriate to the problem at hand.
  - Language design: simplicity, efficiency, generality, etc.
- A PL is *strongly typed* if all type checking can be done at compile time.
- A PL is *type complete* if all objects in the language have equal status.
  - In some languages objects of certain types are restricted.