Chapter 6

Data Types

Topics

- Introduction
- Type Information
- Data Type
  - Specification of a Data Type
- Primitive Data Types
  - Numeric Data Types
    - Integers
    - Floating-Point Numbers
    - Fixed-Point Real Numbers
  - Boolean Types
  - Character Types

Topics

- Composite Data Types
- Character Strings
- User-Defined Ordinal Type
  - Enumerations
  - Subranges
- Structure Types

Topics

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

Introduction

- Every program uses data, either explicitly or implicitly, to arrive at a result.
- All programs specify a set of operations that are to be applied to certain data in a certain sequence.
- Data in its most primitive forms inside a computer is just a collection of bits.

Introduction

- Basic differences among languages exist in the types of data allowed, in the types of operations available, and in the mechanism provided for controlling the sequence in which the operations are applied to the data.
- Most programming languages provide a set of simple data entities as well as mechanism for constructing new data entities from these.

Type Information

- Program data can be classified according to their types.
- Type information can be contained in a program either implicitly or explicitly.

Type Information: implicit

- Implicit type information includes the types of constants and values, types that can be inferred from a name convention, and types can be inferred from context.
  - Example: number 2 is implicitly an integer in most languages
  - Example: TRUE is Boolean
  - Example: variable I in FORTRAN is, in the absence of other information, an integer variable.

Type Information: explicit

- Explicit type information is primarily contained in declarations.
  - Variables can be declared to be of specific types.
  - Example: `var x: array[1..10] of integer;`
  - Example: `var b: boolean;`

Data Type

- A data type is a set of values that can be specified in many ways:
  - Explicitly listed
  - Enumerated
  - Given as a subrange of known values
  - Borrowed from mathematics
- A set of values also has a group of operations that can be applied to the values.
  - These operations are often not mentioned explicitly with the type, but are part of its definition.

Data Type: definition

- A ***data type*** is a set of values, together with a set of operations on those values having certain properties.
- Every language comes with a set of **predefined types** from which all other types are constructed.
  - Provide facility to allow the programmer defined new data types.

Specification of a Data Type

- Basic elements of a specification of a data type:
  1. The *attributes* that distinguish data objects and types.
  2. The *values* that data objects of that type may have, and
  3. The *operations* that defined the possible manipulations of data objects of that type.

Specification of a Data Type: example

- Specification of an array:
  1. The *attributes* might include the number of dimensions, the subscript range for each dimension, and the data type of the components.
  2. The *values* would be the sets of numbers that form valid values for array components.
  3. The *operations* would include subscripting to select individual array components and possibly other operations to create arrays, change their shape, access attributes such as upper and lower bounds of subscripts, and perform arithmetic on pair of arrays.

Primitive Data Types

- Algol-like languages (Pascal, Algol68, C, Modula-2, Ada, C++), all classify types according to a basic scheme, with minor variations.
  - Names used are often different, even though the concepts are the same.
- Primitive types are also called *base types* or *scalar types* or *unstructured types* or *elementary types.*
- A scalar type is a type whose elements consist of indivisible entities (a single data value or attribute).

Numeric Data Types

- Some form of numeric data is found in almost every programming language.
  - Integers
  - Floating-Point Real Numbers
  - Fixed-Point Real Numbers

Integers: specification

- The set of integer values defined for the type forms an ordered subset, within some finite bounds, of the infinite set of integers studied in mathematics.
  - The maximum integer value is sometimes represented as a defined constant
    - Example: In Pascal, the constant *maxint*
    - The range of values is defined to be from *–maxint* to *maxint*
  - Some languages, such as C, have different integer specifications: *short, long*

Integers: specification

- Operations on integer data objects include the main groups:
  - Arithmetic Operations:
    - Binary arithmetic operations such as *addition* (+), *subtraction* (-), *multiplication* (*), *division* (*/* or div), *remainder* (mod).

      BinOp: integer x integer → integer
    - Unary arithmetic operations such as *negation* (-) or *identity* (+).

      UnaryOp: integer → integer

Integers: specification

- Relational Operations:
  - Includes *equal*, *not equal*, *less-than*, *greater-than*, *less-than-or-equal*, and *greater-than-or-equal*.

    RelOp: integer x integer → Boolean
- Assignment

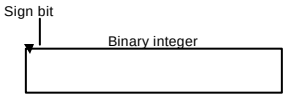
  assignment: integer x integer → integer
- Bit Operations
  - In some languages, integers fulfill many roles.
  - In C, integer also play the role of Boolean values.
  - C includes operations to *and* the bits together (&), *or* the bits together (|), and *shift* the bits (<<).

Integers: implementation

- Implemented using hardware-defined integer storage representation and set of hardware possible storage representations for integers.
- Some possible storage representation for integers.

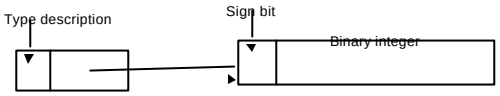
Integers: implementation

- No descriptor
  - Only the value is stored.
  - Possible if the language provides declarations and static type checking for integer data objects.



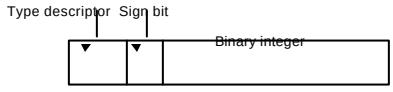
Integers: implementation

- Descriptor separated
  - Stores the description in a separate memory location, with a pointer to the full-word integer value.
  - Often used in LISP.
  - Disadvantage: It may double the storage required.
  - Advantage: because the value is stored using the build-in hardware representation, the hardware arithmetic operations may be used.



Integers: implementation

- Descriptor in the same word
  - Descriptor and value are stored in a single memory location by shortening the size of the integer sufficiently to provide space for the descriptor.
    - Storage is conserved.
    - Hardware operations: clearing the descriptor from the integer data object, perform the arithmetic, and then reinserting the descriptor.
    - Arithmetic is inefficient.



Floating-Point Numbers: specification

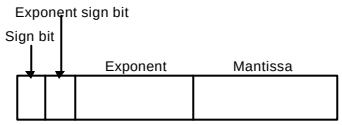
- Sometimes called *real*, as in FORTRAN.
- Also called *float*, as in C.
- As with type integer, the values form an ordered sequence from some hardware-determined minimum negative value to a maximum value, but the values are not distributed evenly across this range.

Floating-Point Numbers: specification

- Same arithmetic, relational, and assignment operations as for integers.
  - Boolean are sometimes restricted.
  - Due to roundoff issues: equality between two real numbers is rarely achieved.
    - Equality may be prohibited by the language designer.
  - Most languages provide other operations as build-in functions:
    - *Sine*: Sin: real → real
    - *Maximum value*: max: real x real → real

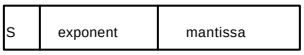
Floating-Point Numbers: implementation

- Storage representation based on an underlying hardware representation.
  - Mantissa (i.e. the significant digits of the number)
  - Exponent



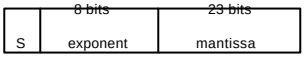
Floating-Point Numbers: implementation

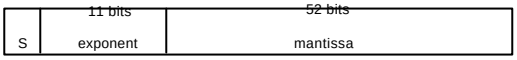
- This model emulates scientific notation
  - Number N can be expressed as $N=mx2^k$
  - IEEE Standard 754
    - 32 and 64 bit standard
  - Numbers consist of 3 fields:
    - S : a one-bit sign field. 0 is positive
    - E: Values (8 bits) range from 0 to 355 corresponding to exponents of 2 that ranges from –127 to 128
    - M: a mantissa of 23 bits. The first number is always 1 then it is inserted automatically by the hardware yielding an extra 24th bit of precision.



Floating-Point Numbers: implementation

- A double-precision form is available in many programming languages.
  - Additional memory word is used to store an extended mantissa.





Fixed-Point Real Numbers: specification

- A fixed-point number is represented as a digit sequence of fixed length, with the decimal point position at a given point between two digits.
  - Example: In COBOL:

    X PICTURE 999V99

    declares X as a fixed-point variable with 3 digits before the decimal and 2 digits after.

Fixed-Point Real Numbers: implementation

- A fixed-point type may be directly supported by the hardware or may be simulated by software.
  - 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 (10,2);

  - Data is stored as *integers*, with the decimal point being an *attribute* of the data object.

Fixed-Point Real Numbers: implementation

- If X has the value 103.421, then r-value of X will be 103 421, and the object X will have an attribute *scale factor (SF)* of three (the decimal point is 3 places to the left).
- The statement Z = X + Y
  - Shift Y left one position (equivalent to multiplying the integral r-value of Y by 10)
  - The sum will have 3 decimal digits (SF=3)
  - Because Z has only 2 decimal places (SF=2) and the sum has 3, remove one place (divide by 10).
- Subtraction and division are handled in an analogous manner.

Boolean Types: specification

- Data objects having one of two values (*true* of *false*).
- The most common operations on Boolean types include assignment and the following logical operations:
  - *and:* Boolean x Boolean → Boolean (conjunction)
  - *or*: Boolean x Boolean → Boolean (inclusive disjunction)
  - *not*: Boolean → Boolean (negation or complement)
- Other Boolean operations: equivalence, exclusive or, implication, nand (not-and), and nor (not-or).

Boolean Types: implementation

- Storage is a single bit providing no descriptor
- Because single bits may not be separately addressable in memory, its storage is extended to be a single addressable unit such as a byte.
- The values *true* and *false* might be represented in 2 ways within this storage unit:
  - A particular bit is used for the value (often, the sign bit of the number representation), with 0=false and 1=true, and the rest of the byte is ignored.
  - A zero value in the entire storage unit represents false, and any other nonzero value represents true.

Character Types: specification

- The set of possible character values is usually taken to be a language-defined enumeration corresponding to the standard character sets supported (I.e. ASCII).
- Operations on character data include only the relational operations, assignments, and sometimes operations to test whether a character is one of the special classes letter, digit, or special character.

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.

Structured Types

- A structured type is a compound type.
  - Arrays, records, tuples, lists, sets, functions, etc.
  - Two kinds of structured types:
    - Heterogeneous: elements of different types.
    - Homogeneous: elements of the same type.

	homogeneous	heterogeneous
static dynamic	array	record

Dynamic selection → homogeneous → compiler will not know which one will be selected at run time

Composite Data Types

- Usually considered elementary data objects.
- Their implementation usually involves a complex data structure organization by the compiler.
- Multiple attributes are often given for each data type.

Character Strings

- Data objects composed of a sequence of characters.
- It is important in most languages
  - Used for data input and output.
- Design Issues
  - Should strings be a special kind of character array or a primitive type (no array-style subscripting operations)?
  - Should string have static or dynamic length?

Character Strings: specification and syntax

- At least 3 different treatments:

1. Fixed declared length.
   - Value assigned: a character string of a certain length.
   - Assignment of a new string value results in a length adjustment of the new string through truncation of excess characters or addition of blank characters to produce a string of the corre4ct length.
   - Storage allocation is determined at translation time.

Character Strings: specification and syntax

2. Variable length to a declared bound.
   - The string may have a *maximum* length that is declared previously.
   - The actual value stored may be a string of shorter length (even the empty string).
   - During execution, the length of the string value may vary, but it is truncated if it exceeds the bound.
   - Storage allocation is determined at translation time.
3. Unbound length.
   - The string may have a string value of any length.
   - The length may vary dynamically during execution with no bound (beyond available memory).
   - Dynamic storage allocation at run time.

Character Strings: C

- Strings are arrays of characters (no string declaration).
- Convention: null character ("\0") follows the last character of a string.
- Every string, when stored in an array, will have the null character appended by the C translator.
  - Programmers have to manually include the final null character to strings made from programmer-defined arrays.

Character Strings: operations

- A wide variety of operations are usually provided.
  1. *Concatenation.*
     - Operation of joining two strings to make one long string
     - Example: if || is the symbol used for concatenation, "BLOCK" || "HEAD" gives "BLOCKHEAD"

Character Strings: operations

2. Relational operations on strings.

- Usual relational operations (equal, less-than, greater-than, etc) may be extended to strings.
- Lexicographic (alphabetic) order
  - Example: String A is less than String B if either
    - The first character of A is less than the first character of B
    - If both characters are equal and the second character of A is less than the second character of B, and so on.
    - A shorter string is extended with blank character (spaces) to the length of the longer.

Character Strings: operations

3. *Substring selection using positioning subscripts.*
   - Some languages provide an operation for selecting a substring by giving the position of its first and last characters
     - Or first character position and length of the substring.
     - Example: In Fortran: `Next = STR(6:10)`
   - Some problem could arise if substring selection appears on both sides of an assignment
     - Example: In Fortran, `STR(!:5) = STR(I:I+4)`

Character Strings: operations

4. *Input-output formatting.*
   - Formatting data for output.
   - Breaking up formatted input data into smaller data items.
5. *Substring selection using pattern matching.*
   - Often the position of a desired substring within a larger string is not know.
     - Its relation to other substrings is know.
     - Examples:
       - A sequence of digits followed by a decimal point
       - The word following the word THE.

Character Strings: operations

- *Patter matching operation* takes two arguments:
  - A pattern data structure
    - The pattern specifies the form of the substring desired and possibly other substrings that should adjoin it.
    - A string with a substring that matches the specified pattern.
  - The most common pattern matching mechanism are **regular expressions.**
  - Some languages have pattern matching built into the language (Perl, Python, Ruby, …).
  - Some languages implement pattern matching via external libraries or classes
    - Java has Pattern and Matcher classes

Recursive Definition of a Regular Expression

- Individual terminals are regular expressions
- If a and b are regular expressions so are
  - a | b choice
  - ab sequence
  - (a) grouping
  - a* zero or more repetitions
- Nothing else is a regular expression

Examples

- Identifiers
  - letter(letter | digit)*
- Binary strings
  - (0 | 1)(0 | 1)*
- Binary strings divisible by 2
  - (0 | 1)*0

Pattern Symbols

Symbol	Meaning
.	Any single character (except '\n')
*	0 or more occurrences
+	1 or more occurrences
?	0 or 1 occurrences of previous character
[abc]	One of enclosed characters
[^abc]	None of enclosed characters
{i, j}	Between i and j occurrences
|	Choice
()	Grouping
\i	Case insensitive

Character Classes

- There are several classes of characters that have special names

Match		Exclude
\d	Any digit	\D
\w	Any letter, digit, or underscore	\W
\s	Any whitespace	\S

Anchors

- Used to specify position within a string

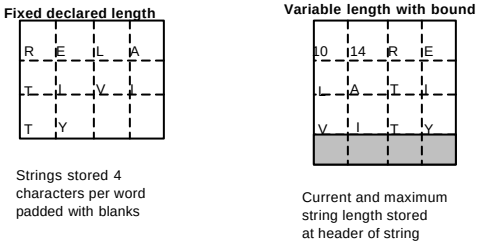
Symbol	Position
^	Beginning of string
$	End of string
\b	Word boundary
\B	Not at word boundary

- \bpattern\b matches the word pattern but not patterned

Character Strings: implementation

- Each of the 3 methods for handling character string utilizes a different storage representation.
- Hardware support for the simple fixed-length representation is usually available but other representations for strings must usually be software simulated.

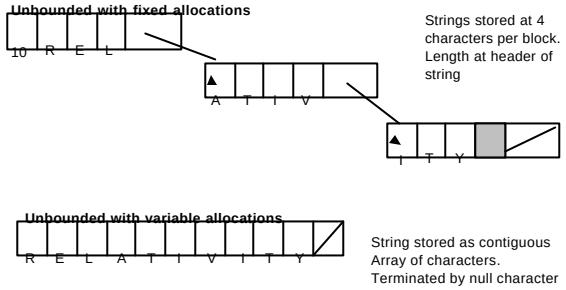
Storage Representation for Strings



Strings stored 4 characters per word padded with blanks

Current and maximum string length stored at header of string

Storage Representation for Strings



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

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

Strings: evaluation

- String types are important to writability and readability.
- Not costly (language or compiler complexity): add strings as a primitive type.
  - Standard libraries is as convenient as using strings as primitive types.

User-Defined Ordinal Type

- **Ordinal type**: the range of possible values can be easily associated with the set of positive integers.
  - Enumerations
  - Subrange

Enumerations

- A variable can take only one of a small number of symbolic values.
  - Example: a variable *StudentClass* might have only four possible values representing freshman, sophomore, junior, and senior.
  - Example: a variable EmployeeSex might have only two values representing male and female.
  - In older languages (Fortran, Cobol), an enumeration variable is given the data type integer.
    - The values are represented as distinct, arbitrary chosen integers.
    - Example: Freshman = 1, Sophomore = 2, and so on.
    - Example: Male = 0, Female = 1

Enumerations: specification

- Modern languages include an enumeration data 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 using a declaration.
  - Example: In C,

```
enum StudentClass {Fresh, Soph, Junior, Senior};
enum EmployeeSex {Male, Female};
```


Enumerations: specification

- Ordinarily many variables of the same enumeration type are used in a program.
- Define the enumeration in a separate type definition
  - In Pascal: **type** Class = {Fresh, Soph, Junior, Senior};
    followed by declarations for variables:

```
StudentClass: Class; TransferStudent: Class;
```

  - The type definition introduces the type name Class, which may be used as a primitive.

Enumerations: specification

- It also introduces the *literals* of Fresh, Soph, Junior, and Senior which may be used instead of the corresponding integers.
  - Example: **if** StudentClass = Junior **then** …

    instead of the less undertandable

    if StudentClass = 3 **then** …
- Static type checking by the compiler could find programming errors.
  - Example: **if** StudentClass = Male **then** …

Enumerations: operations

- Basic operations
  - Relational operations (equal, less-than, etc)
    - Defined for enumerations types because the set of values is given an ordering in the type definition.
  - Assignment
  - Successor and predecessor
    - Gives the next and previous value, respectively, in the sequence of literals defining the enumeration.
    - Undefined for the last and first values, respectively

Enumerations: implementation

- Storage representation is straightforward
  - Each value in the enumeration sequence is represented at run time by one of the integer 0, 1, ….
  - Only a small set is involved and the values are never negative.
  - The usual integer representation is often shortened to omit the sign bit and use only enough bits for the range of values required.

Enumerations: implementation

- Example: The previous type Class has only four possible values 0 = Fresh, 1 = Soph, 2 = Junior, and 3 = Senior.
- Only 2 bits are required to represent these 4 possible values in memory.
- Successor & predecessor: add or subtracts one form the integer representing the value and check if the result is within the proper range.

- In C, the programmer may override this default and set any values desired.
  - Example:

```
enum class {Fresh=14, Soph=36, Junior=4, Senior=42};
```


Enumerations: evaluation

- Provides advantages in
  - Readability
    - Named values are easily recognized.
  - Reliability
    - In languages such as C#, Ada, Java there are two advantages:
      - No arithmetic operations are legal on enumeration types.
      - No enumeration variable can be assigned a value outside its defined range.
    - In C there are no advantages (treats enumerations as integers).

Subrange Types: specification

- A subrange is a contiguous sequence of an ordinal type within some restricted range.
  - Example: In Pascal, `A: 1..10`
    In Ada, `A: integer range 1..10`
- A subrange type allows the same set of operations to be used as for the corresponding ordinal type.

Subrange Types: implementation

- Two important effects on implementations:
  1. *Smaller storage requirements.*
     - Because a smaller range of values is possible, a subrange value can usually be stored in fewer bits than a general integer value.
       - Example: the subrange `1..10` requires only 4 bits whereas a full integer requires 16 or 32.
       - Because arithmetic operations on shortened integers may need software simulation for their execution (slower), subranges values are often represented as the smallest number of bits for which the hardware implements arithmetic operations (generally 8 or 16).

Subrange Types: implementation

2. *Better type checking.*
   - More precise type checking to be performed on the values assigned to the variable.
   - Example: if variable `Month` is `Month: 1..12`, then the assignment `Month := 0` is invalid and can be detected at compile time. If `Month` is declared to be of integer type, then the assignment is valid and the error must be found by the programmer during testing.
   - Some subrange type checks cannot be performed at compile time, i.e in `Month := Month + 1` run time checking is needed to determine whether the new value is within the bounds declared.

Subrange Types: evaluation

- Enhance readability by showing clearly that variables of subtypes can store only certain ranges of values.
- Increase reliability because possible values that are outside of a range can be detected faster and easier.
- No contemporary language except Ada95 has subrange types.

Structured Types

- A structured type is a compound type that contains other data objects as its elements or components.
  - Arrays
  - Records
  - Tuples
  - Lists
  - Sets
  - Functions
  - Stacks

Structure Types: specification

- The major attributes for specifying structure types include:

1. *Number of components*
   - *Fixed size:* the number of components is invariant during its lifetime.
     - Arrays and records.
   - *Variable size:* the number of components changes dynamically.
     - Usually define operations that insert and delete components.
     - Stacks, lists, sets, tables, and files.

Structure Types: specification

2. *Type of each component*
   - Two kinds of structured types:
     - Heterogeneous: elements of different types.
     - Homogeneous: elements of the same type.

	homogeneous	heterogeneous
static		record, list
dynamic	array, set, file	

Dynamic selection → homogeneous → compiler will not know which one will be selected at run time

Structure Types: specification

3. *Names to be used for selecting components.*
   - Needs a selection mechanism for identifying individual components of the data structure
     - Array: name of an individual component may be an integer subscript or sequence of subscripts.
     - Table: the name may be a programmer-defined identifier.
     - Record: name is usually a programmer-defined identifier.
     - Some data structures (stacks and files) allow access to only a particular component (top or current component) at any time.

Structure Types: specification

4. *Maximum number of components.*
   - For variable-size data structures, a maximum size for the structure in terms of number of components may be specified.
5. *Organization of the components.*
   - The most common organization is a simple linear sequence of components.
     - Vectors (one-dimensional arrays), records, stacks, lists, and files.
     - Array, record, and list types are usually extended to multidimensional forms: multidimensional arrays, records whose components are records, lists whose components are lists.

Structure Types: operations

1. *Component selection operations.*
   - Processing data often proceeds by retrieving each component of the structure.
   - Two types of *selection operations*:
     - *Random selection*: an arbitrary component of the data structure is accessed.
     - *Sequential selection*: components are selected in a predetermined order.

Structure Types: operations

2. *Whole-data structure operations.*
   - Operations may take entire data structures as arguments and produce new data structures as results.
   - Most languages provide a limited set.
     - Addition of two arrays.
     - Assignment of one record to another.
     - Union operation on sets.

Structure Types: operations

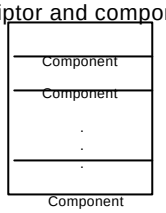
3. *Insertion/deletion of components.*
   - Operations that change the number of components in a data structure.
4. *Creation/destruction of data structures*
   - Operations that create and destroy data structures.

Structure Types: implementation

- The storage representation includes (1) storage for the components of the structure, and (2) an optional descriptor that store some or all of the attributes of the structure.
- There are two basic representations:
  - Sequential
  - Linked

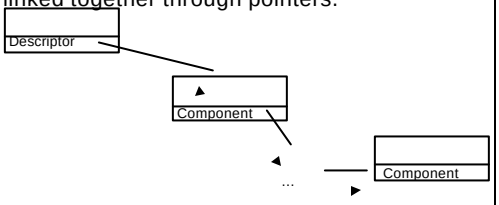
Structure Types: implementation

1. *Sequential representation.*
   - The data structure is stored in a single contiguous block of storage that includes both descriptor and components.



Structure Types: implementation

2. *Linked representation.*
   - The data structure is stored in several non-contiguous blocks of storage, with the blocks linked together through pointers.



Structure Types: implementation

- Sequential representations are used for fixed-size structures and sometimes for homogeneous variable-size structures such as character strings or stacks.
- Linked representations are commonly used for variable-sized structures such as lists.

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]

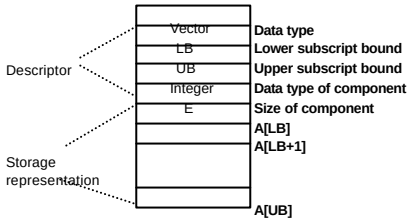
Vectors: other operations

- 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 $(I-1) \times E$ memory locations.
  - If LB is the lower bound on the subscript range, then the number of such components to skip is $I-LB$ or $(I-LB) \times E$ 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

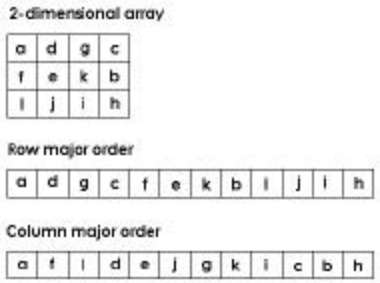
Arrays: implementation

- 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)*

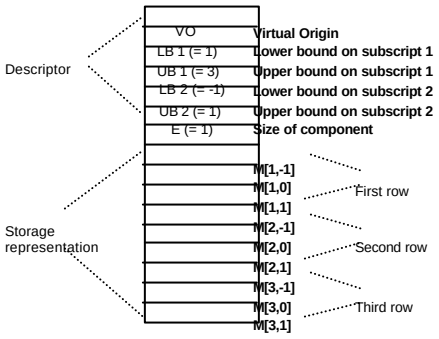
Arrays: Row- vs. Column-major order



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

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

where $S = \text{length of a row} = (UB_2 - LB_2 + 1) \times E$

  - A is stored in column-major order

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

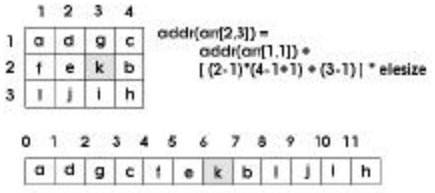
where $S = \text{length of a row} = (UB_1 - LB_1 + 1) \times E$

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

location(A[I,J]) = α + (I - LB_1) x S + (J - LB_2) x E

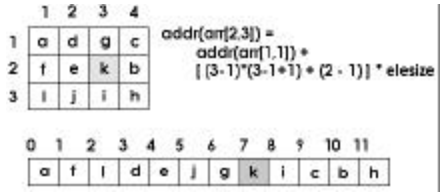
S = (UB_2 - LB_2 + 1) x E



Arrays: Column-major access function

location(A[I,J]) = α + (J - LB_2) x S + (I - LB_1) x E

S = length of a row = (UB_1 - LB_1 + 1) x 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.

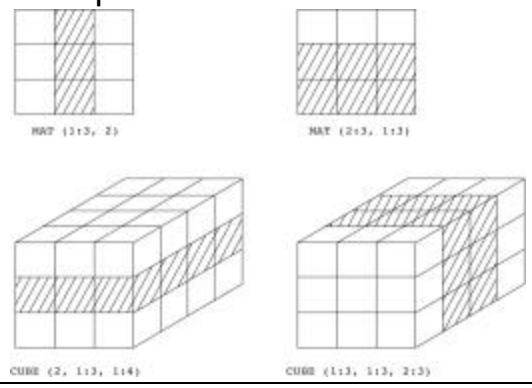
1. In FORTRAN 90

   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



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};
```

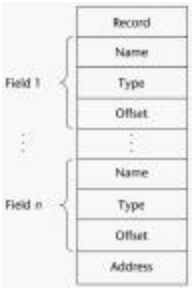
- Define & initialize using the dot operator (structure member operator)

```
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.

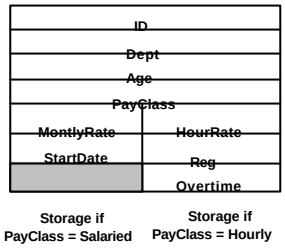
Unions: type checking

- 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.

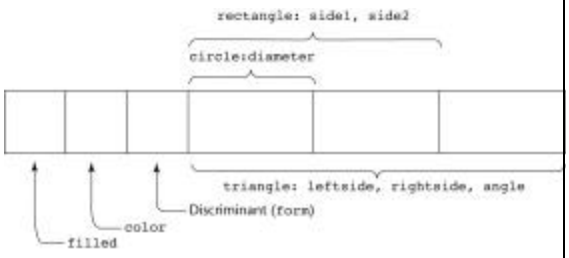
Unions: implementation

- 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



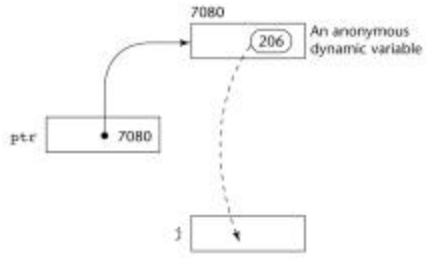
Union: evaluation

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

Pointers

- A pointer type is a type in which the range of values consists of memory addresses and a special value (nil, or null)
- Pointers are useful for
  - Addressing flexibility
  - Dynamic storage management
- Pointer operations
  - Assignment of an address to a pointer
    int *p, x = 5;
    p = &x;
  - Dereferencing
    ***p = 12;**

Pointers



The assignment operation j = *ptr

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.