Chapter 6

Data Types

Topics

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

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(I: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

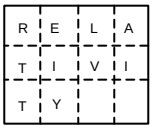
- \bpattern\b matches the word pattern but not patterned

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.

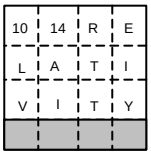
Storage Representation for Strings

Fixed declared length



Strings stored 4 characters per word padded with blanks

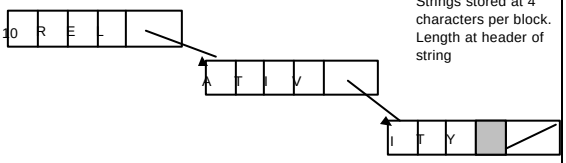
Variable length with bound



Current and maximum string length stored at header of string

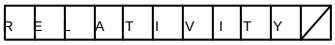
Storage Representation for Strings

Unbounded with fixed allocations



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

Unbounded with variable allocations



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

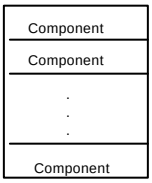
3. *Insertion/deletion of components.*
   - Operations that change the number of components in a data structure.
4. *Creation/destruction of data structures*
   - Operations that create and destroy data structures.

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

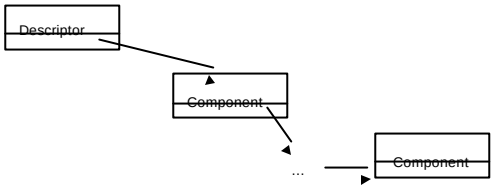
Structure Types: implementation

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.