

Ch. 5: Types

I. Introduction

- A. A program can be evaluated on how well its data-types match the real-world problem space
- B. Early languages had only a few non-extensible basic data structures
- C. Steps toward allowing flexible data structures:
 - 1. COBOL allowed specifying accuracy of decimal data; records
 - 2. ALGOL 68: provided a few basic types and a few flexible structure-defining operators
- D. Ada: extended this idea of user-defined types. The design philosophy is that the user should be allowed to create a unique type for each unique class of variables.
- E. Fundamental abstract data type idea: the use of a type is separate from the representation and set of operations.
- F. We can think of variables in terms of descriptors, where a descriptor stores the attributes of a variable.
 - 1. For static variables, descriptors needed only at compile time
 - 2. For dynamic variables, descriptors needed at run time.

II. Primitive Data Type: A type that is not defined in terms of any other type

A. Numeric Types

- 1. Integer
 - a. Often different sizes: short (byte), int (word), long (2 words), quad (4 words)
 - b. Representations may be unsigned, signed-magnitude, or twos-complement
- 2. Floating-Point
 - a. Base-2 approximations of decimal numbers
 - b. Different sizes: real, double
 - c. Can be emulated in software (10 - 100 times slower)
- 3. Decimal
 - a. Precisely store decimal values within a restricted range
 - b. BCD - Binary Coded Decimal
 - c. One or two digits per byte: It takes at least 4 bits per digit (Compare space required for large numbers, $> 2^{20}$)

B. Boolean Type

- 1. Two values used to represent True & False
- 2. In C: 0 & 1
- 3. Could be represented by 1 bit, though machines usually only byte-addressable.

C. Character Types: ASCII

III. Character String Types

A. Design Issues

- 1. Should strings be primitives or simply an array of chars?
- 2. Should they have static or dynamic length

B. Strings & their Operations

- 1. Arrays of chars. used in Pascal, C, C++, & Ada.
- 2. Ada: substring references (e.g. StrName(2:4)), catenation (str3 := str1 & str2)
- 3. C & C++: strings terminated w/null character. Library functions to copy, catenate, compare, get length
- 4. FORTRAN 77 & 90, BASIC have strings as a primitive type

C. String Length Options

1. static length string (e.g. In Pascal: ARRAY [1..10] OF INTEGER)
2. limited dynamic length strings (e.g. in C: char *text = "Max size text";, where the string could be less than the max declared length)
3. Dynamic length strings: dynamically allocated & deallocated

D. Implementation of Character String Types

1. Descriptor for static strings must include: static string, length, address
2. Descriptor for dynamic strings must include: dynamic string, max. length, current length, address

IV. User-Defined Ordinal Types: An ordinal type is one in which the range of values can easily associated with the set of positive integers (e.g. Pascal's integer, char, & boolean)

A. Enumeration Types: all the possible values (symbolic constants) are enumerated in the definition. Issue: is a literal constant allowed to appear in more than one type definition?

1. Designs

- a. Pascal: literal constant can only appear once

```
TYPE colorType = (red, blue, green, yellow);
VAR color: colorType;

...
color := blue;
if (color > red) ...
```

- b. Ada: allows overloaded literals

```
TYPE Letters IS ('A', 'B', 'C', 'D', 'E', ... 'Z'); {all must actually be listed}
TYPE Vowels IS ('A', 'E', 'I', 'O', 'U');
Then using:
FOR letter IN 'A'..'U' LOOP
Which type should be used? This can be disambiguated using:
FOR letter IN Vowels ('A')..Vowels('U') LOOP
```

2. Evaluation: Enumerated types enhance readability. The alternative (e.g. FORTRAN) is coding enumerated values as integers, where then the compiler would not be able to detect out of range values.

B. Subrange Types: a contiguous subsequence of an ordinal type, e.g. 3..5 is a subset of integers. Examples:

1. Pascal

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

2. Ada

```
SUBTYPE Weekdays IS Days RANGE Mon..Fri;
SUBTYPE Index IS Integer RANGE 1..100;
```

Note that in Ada subtypes are not new types, but rather constrained versions of existing types.

C. Implementation: subrange types are implemented as their parents, with the added overhead of range checks on every assignment.

V. Array Types: An array is a homogeneous aggregate of data elements in which the individual element is identified by its position relative to the first element. Elements are of some previously defined type.

A. Design Issues

1. What types are legal for subscripts?
2. When are subscript ranges bound?
3. When does array allocation take place?
4. How many subscripts are allowed?
5. Can they be initialized when allocated?
6. What kind of slices, if any, are allowed?

B. Arrays and Indexes

1. Referencing an element is done by a two-level mechanism
 - a. Aggregate name is used
 - b. Dynamic or Static selector mapping from the array name & index values to an individual element
e.g. this mapping is `array_name[index] -> element`
2. Parenthesis or Brackets may be used. Problem w/ parenthesis
e.g. FORTRAN `SUM = SUM + B(I)`
Is this an array reference or a function call? In FORTRAN external functions need not be declared, so an erroneous function call may not be caught.
3. Ada: deliberately chose parenthesis for use both in arrays & function calls, as both can be seen as mappings

C. Subscript Bindings and Array Categories

1. Static: subscript ranges and storage allocation are both static, e.g. FORTRAN 77
2. Fixed Stack-Dynamic: subscript ranges are statically bound, but the allocation is done at elaboration time on the stack during execution. e.g. in C

```
int myFunction () {  
    int x[5]; ...  
}
```

3. Stack-Dynamic: subscript ranges and storage allocation are dynamically bound. Sizes could be different each time. e.g. in Ada:

```
GET (List_Len);  
DECLARE  
    List: ARRAY (1..List_Len) OF INTEGER;  
BEGIN  
    ...  
END;
```

4. Heap-Dynamic: binding of subscript ranges and storage allocation is dynamic & can change over the course of the array's lifetime. E.g. FORTRAN 90:

```
INTEGER, ALLOCATABLE, ARRAY (:,:) :: Matrix  
and then storage might later be allocated with  
    ALLOCATE (Matrix(10, NumberOfColumns))  
and later destroyed using  
    DEALLOCATE (Matrix)
```

C can also allocate arrays dynamically using `malloc`

D. The Number of Subscripts in Arrays

1. Original FORTRAN: 3, FORTRAN IV: 7. (0,1, infinity principle)
2. Arrays are not a basic data type in C, since there are no operations for them. Multi-dimensioned arrays are actually arrays of arrays, e.g.: `int Matrix [5][4];`

E. Array Initialization

1. In FORTRAN:

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

2. In C:

```
int list [] = {1, 3, 5, 7, 11}
where the compiler sets the length of the array
```

3. In Ada a collection of values called aggregate values can be initialized:

```
List: ARRAY (1..5) OF INTEGER := (1, 3, 5, 7, 9);
Bunch: ARRAY (1..5) OF INTEGER := (1 => 3, 3 => 4, OTHERS => 0);
```

F. Array Operations: e.g. APL

G. Slices

Given FORTRAN 90 declarations:

```
INTEGER Vector(1:10), Mat(1:3, 1:3), Cube(1:3, 1:3, 1:4)
```

We can access certain groupings only (a slice), as illustrated in [Fig. 5.4]

H. Implementation of Array Types

1. Access functions

a. For example take

```
list[k]
```

where the access function for list would be

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

This illustrates that every array dimension necessitates one add and one multiply instruction for the access function.

b. The descriptor could look like: Array, Element type, Index type, Number of dimensions, Index 1 bounds, Index 2 bounds, ... Index n bounds, Address

2. Row-major vs. Column-major: storage order can influence system efficiency due to paging by the O.S. E.g.

Consider the 4 x 4 matrix called myMatrix:

```
1 3 5 7
2 4 6 8
3 7 11 14
4 10 16 21
```

Assuming the OS has only 1 page for the array, and that the page can only hold 4 data items at a time, and that the array is stored row-major, then in Pascal:

```
FOR i:= 1 TO 4 DO
  FOR j:= 1 TO 4 DO
    WRITE( myMatrix[ i, j]);    { 4 page faults }
```

but flipping the indices to vary i inside j:

```
FOR j:= 1 TO 4 DO
  FOR i:= 1 TO 4 DO
    WRITE( myMatrix[ i, j]);    { 16 page faults }
```

VI. Record Types

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names. E.g. fields for customer record.

- Records introduced in early 60's by COBOL

A. Structure of records: difference between records and arrays is that array elements are homogeneous, and record elements need not be. Record fields are named and referenced with identifiers.

1. E.g. [COBOL record declaration] level numbers, format & length specifiers.

2. Pascal, Ada use [nested record declarations]

3. C: structures

4. FORTRAN 90: nested records must be previously defined types

B. References to Record fields.

1. Two main approaches:

a. COBOL field references: field1 OF record1 OF... OF record_n, where the first field name is at the innermost level, and the last is the outermost.

E.g. MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE -RECORD

b. dot notation: opposite order than COBOLS, with largest enclosing record occurring first.

E.g. In Ada: EMPLOYEE_RECORD.EMPLOYEE_NAME.MIDDLE

2. Fully qualified reference to a record field has all intermediate record names

3. Elliptical references names fields, but leaves out unambiguous intermediate record names. E.g from the COBOL example: FIRST, FIRST OF EMPLOYEE_NAME, and FIRST OF EMPLOYEE-RECORD would all be the same.

4. Pascal's WITH clause [overhead]

C. Operations on records

1. Pascal: records can be assigned but nothing else

2. Ada: assignment & comparison of records; overloading can provide further record operations

3. ANSI C & C++: records can be assigned

4. COBOL's MOVE CORRESPONDING allows copying fields with same names into different-structured record [overhead]

D. Implementation of Record Types: [overhead]

1. Records are like arrays with constant subscripts

2. Corresponding assembler instruction format: (d, An, Ri), which is address indirect with displacement and Index Register.

VII. Union Types

A union is a type that may store different type values at different times during program execution

A. FORTRAN Union Types. E.g.:

INTEGER X

REAL Y

EQUIVALENCE (X, Y)

where X and Y are aliases for the same storage location. No type checking is done.

B. Algol 68 Union Types: wanted to be able to check the current type value in a union.

1. discriminated union, where a tag (discriminant) identifies the current union type.

E.g. ALGOL 68:

```
union (int, real) ir1, ir2
```

where ir1 and ir2 can be either INT or REAL type values. Referencing can be tricky:

```
UNION (int, real) ir1;
```

```
int count;
```

```
...
```

```
ir1:= 33;// Legal: type of ir1 must be int
```

```
...
```

```
count := ir1;// Illegal, since compiler can't statically check type of ir1
```

To avoid this problem in ALGOL 68, conformity clauses are added (like casts in c):

```
UNION (int, real) ir1;
```

```
int count;
```

```
real sum;
```

```
...
```

```
case ir1 in
```

```
    (int intval): count := intval,
```

```
    (real realval): sum := realval
```

```
esac
```

where "intval" and "realval" are dummy variables whose scope is only the statement following their specifications

C. Pascal Union types: record variant [overhead of code & storage]

1. To print the variant [overhead]

2. Problem: a program can change the tag without changing the variant, so tag can simply be omitted, giving a free union [overhead], which can have incorrect references

3. Pascal variant records can be used to use a single storage location to be treated both as pointers (addresses) and integers, allowing the otherwise prohibited pointer arithmetic.

D. Ada Union types: safer than Pascal version: tag cannot be changed without the variant being changed, and tag is always required on variant records. [overhead]

1. FIGURE_1 is an unconstrained variant record whose type can change by assignment of a whole new record [overhead]

2. FIGURE_2 cannot be changed to another variant

E. C & C++ union types are free unions

F. Implementation of Union types: [overhead]

VIII. Set Types

A set type is one whose variables can store unordered collections of distinct values from some ordinal type called its base type.

A. Pascal & Modula-2 are the only ones with a set type

B. Pascal's set type: implementation dependent, and often severely restricted in size

1. Example [overhead]

2. Provides for set union, intersection, and equality

C. Ada does not include set types, though it has a set membership operator for its enumeration types

D. Set operations can always be implemented explicitly by the programmer, though it can be inconvenient

E. Implementation of set types: Usually done as bit strings in memory, e.g. ['a', 'c', 'h', 'o'] could be represented as:

```
1010000100000010
```

which then allows using single machine instructions (logical operations) such as AND and OR

IX. Pointer Types

- *Pointers* are variables that can store memory addresses or a special value called *nil*.
- 2 Uses: indirect addressing, and dynamic storage management using the heap
- *Dynamic variables* are those dynamically allocated from the heap. They often do not have names and can be referenced only by pointer variables and so are called *anonymous variables*

A. Design Issues

1. What are the scope & lifetime of a pointer variable?
2. What is the lifetime of a dynamic variable?
3. Are pointers restricted as to the type of object to which they can point?
4. Are pointers used for dynamic storage management? Indirect addressing?

B. Pointer Operations: assignment & dereferencing

1. *Assignment*: set a pointer variable to the address of some object
2. *Dereferencing*: following one level of indirect addressing (the pointer stores the address of the object, rather than the object itself).
 - a. E.g. in C

```
int *numPtr;  
int num = 5;  
numPtr = &num;
```

- b. E.g. in Pascal

```
num2 := numPtr^; { num2 gets contents of address that numPtr points to }
```

3. Referencing records

- a. In C: If p points to a record with field age, then (*p).age refers to that field. A shortcut notation for the same thing is p->age, which combines dereference and field reference
- b. In Pascal: p^.age
- c. In Ada: p.age, as p is implicitly dereferenced.

C. Problems: PL/I as an example

1. Type Checking: if a pointer's type is not restricted, it is not possible to do static type checking
2. Dangling References: a pointer containing the address of a dynamic variable that has been deallocated.

- a. Example in PL/I:

```
BEGIN;  
DECLARE POINT POINTER;
```

```
...
```

```
    BEGIN;  
    DECLARE SUM FIXED;  
    POINT = ADDR(SUM);
```

```
    ...
```

```
    END;
```

```
... { pointer POINT still exists, but the object it pointed to has been deallocated }  
END;
```

b. The deallocation might also be done explicitly, as in:

```
{ "Based" means it can be allocated & freed, POINT is a pointer to it. }  
DECLARE SUM FLOAT BASED (POINT);  
DECLARE POINT2 POINTER;  
...  
ALLOCATE SUM;  
POINT2 = POINT;  
FREE SUM; { POINT is now a dangling pointer due to the explicit deallocation }
```

3. Lost Objects: An object that is no longer accessible, but may still contain useful data. E.g.

```
DECLARE SUM FLOAT BASED (POINT);  
...  
ALLOCATE SUM;  
...  
ALLOCATE SUM; {First SUM still exists, but is no longer accessible}
```

D. Pointers in Pascal:

1. Used to access dynamically allocated anonymous variables. Uses NEW and DISPOSE.
2. Dangling pointers are a problem.
3. No reference checking or garbage collection done.

E. Pointers in Ada: similar to Pascal with the following enhancements:

1. A dynamic variable may be implicitly deallocated at the end of the scope of its pointer type, leaving no pointers pointing at that object.
2. All pointers are implicitly initialized to NULL (Ada's version of nil)

F. Pointers in C and C++:

- Used extensively, very powerful, but also very dangerous.
- C uses "*" for dereferencing, and "&" to get the address of a variable. E.g.

```
int *ptr;  
int count, init;  
...  
ptr = &init;  
count = *ptr; { same as if we had done count = init; }
```

- Pointer arithmetic scales the increment by the size of the object. E.g.

```
ptr + index { if *ptr is an int, index advances by one int. If long... long }
```

- Arrays use 0 as the first subscript, and array names without subscripts refer to the address of the first element, so given:

```
int list [10];  
int *ptr;  
ptr = list;
```

then

```
*(ptr + 1) is equivalent to list [1];  
*(ptr + index) is equivalent to list[ index]  
ptr[0] is equivalent to list[0]
```

- Note that pointer operations include the same scaling that is used in array indexing operations

G. Pointers in FORTRAN 90: since pointer dereferencing is implicit, a special assignment statement (pointer => target) is used when dereferencing isn't desired.

H. C++ Reference Types

1. A reference type variable is a constant pointer that is always implicitly dereferenced. This implies that when used as a parameter, it is initialized with some address in its definition and that address can not change (but the contents at that address can)
2. E.g. of switching two numbers in C++

I. Implementation of Pointer Types

1. Representation of Pointers: could be an int, or on Intel it is a segment & an offset
2. Solutions to the dangling pointer problem
 - a. Tombstones: All pointers point to a special cell called a tombstone, which in turn points to the dynamic variable itself. When the dynamic variable is deallocated, the tombstone remains but is marked as nil. [Fig. 5.12]
 - b. Locks-and-key: pointer values are ordered pairs (key, address) where the key is an integer value. Dynamic variables also have a header storing a corresponding lock. Every reference verifies the lock and key match. Once the dynamic variable is deallocated, other dangling references will not have a matching key and so will give a run-time error
3. Heap Management:
 - This is a problem particularly for languages like LISP where the entire programs and data consist of linked lists.
 - Allocation is easy: simply find some free space
 - Deallocation is much harder: what if multiple pointers point to same object?
 - a. Reference counters (eager approach)
 - maintain counters in every cell (object). When it goes down to zero, deallocate
 - Problems: 1. space overhead 2. execution time overhead, 3. Circular connections that are not externally accessible are not deallocated
 - b. Garbage collection (aka lazy approach)
 - Don't do anything until you run out of space. Then mark everything as garbage. Then access every cell pointed to in the program and mark it to keep it. Deallocate everything else. [see overhead for marking algorithm code & tree]
 - Problem with this approach: when you need it most (little space left) it works the worst (takes longest to mark many valid pointers)
 - c. Problem is much more difficult when variable-size cells are allocated. This is usually the case in imperative languages.
 - E.g. for garbage collection, how do you tell what a "cell" is in order to mark it? Also, maintaining the list of free space is complicated due to fragmentation.