Data Types

Chapter 5

Introduction

This Chapter introduces the concept of a data type and discusses:

- Characteristics of the common primitive data types.
- Character strings
- User defined data-types
- Design of enumerations and sub-range data types
- Design of structured data types including arrays, records, unions and set types.
- Pointers and heap management

Data Types

- Every PL needs a variety of data types in order to better model/match the world
- More data types makes programming easier but too many data types might be confusing
- Which data types are most common? Which data types are necessary? Which data types are uncommon yet useful?
- How are data types implemented in the PL?

Evolution of Data Types

FORTRAN I (1956) - INTEGER, REAL, arrays

Ada (1983) - User can create a unique type for every category of variables in the problem space and have the system enforce the types

Def: A *descriptor* is the collection of the attributes of a variable

Design Issues for all data types:

1. What is the syntax of references to variables?
2. What operations are defined and how are they specified?

Primitive Data Types

These types are supported directly in the hardware of the machine and not defined in terms of other types. E.g.:

- **Integer:** Short Int, Integer, Long Int (etc)
- **Floating Point:** Real, Double Precision

  Stored in 3 parts, sign bit, exponent and mantissa (see fig 5.1 page 199)
- **Decimal:** BCD (1 digit per 1/2 byte)

  used in business languages with a set decimal for dollars and cents
- **Boolean:** (TRUE/FALSE, 1/0, T/NIL)
- **Character:** Using EBCDIC, ASCII, UNICODE, etc.

Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada)

```
type SPEED is digits 7 range 0.0..1000.0;
type VOLTAGE is delta 0.1 range -12.0..24.0;
```

- IEEE Floating Point Standard 754
  - Single precision: 32 bit representation with 1 bit sign, 8 bit exponent, 23 bit mantissa
  - Double precision: 64 bit representation with 1 bit sign, 11 bit exponent, 52 bit mantissa

Decimal and Boolean

Decimal

- For business applications (money)
- Store a fixed number of decimal digits (coded)
- *Advantage:* accuracy
- *Disadvantages:* limited range, wastes memory

Boolean

- Could be implemented as bits, but often as bytes
- *Advantage:* readability

Character Strings

- Characters are another primitive data type which map easily into integers.
- We've evolved through several basic encodings for characters:
  - 50s – 70s: EBCDIC (Extended Binary Coded Decimal Interchange Code) -- Used five bits to represent characters
  - 70s – 00s: ASCII (American Standard Code for Information Interchange) -- Uses seven bits to represent 128 possible "characters"
  - 00s - : Unicode -- Uses 16 bits to represent ~64K different characters

    Needed as computers become less eurocentric to represent the full range of non-roman alphabets and pictographs.

Character String Types

Values are sequences of characters
Design issues:
- Is it a primitive type or just a special kind of array?
- Is the length of objects static or dynamic?

Typical String Operations:
- Assignment
- Comparison (=, >, etc.)
- Catenation
- Substring reference
- Pattern matching

Character Strings

- Should a string be a primitive or be definable as an array of chars?
  - In Pascal, C/C++, Ada, strings are not primitives but can "act" as primitives if specified as "packed" arrays (i.e. direct assignment, <, =, > comparisons, etc...).
  - In Java, strings are objects and have methods to support string operations (e.g. length, <, >)
- Should strings have static or dynamic length?
- Can be accessed using indices (like arrays)
- Operations: comparison, assign, input/output, length, concatenation, append, substr, etc...

String examples

- SNOBOL - had elaborate pattern matching
- FORTRAN 77/90, COBOL, Ada - static length strings
- PL/I, Pascal - variable length with static fixed size strings
- SNOBOL, LISP - dynamic lengths
- Java - objects which are immutable (to change the length, you have to create a new string object) and + is the only overloaded operator for string (concat), no overloading for <, >, etc

String Examples

- Some languages, e.g. Snobol, Perl and Tcl, have extensive built-in support for strings and operations on strings.
- SNOBOL4 (a string manipulation language)
  - Primitive data type with many operations, including elaborate pattern matching
- Perl
  - Patterns are defined in terms of regular expressions providing a very powerful facility!
    /[A-Za-z][A-Za-z\d]+/
- Java - String class (not arrays of char)

String Length Options

Static - FORTRAN 77, Ada, COBOL
e.g. (FORTRAN 90)
CHARACTER (LEN = 15) NAME;

Limited Dynamic Length - C and C++ actual length is indicated by a null character

Dynamic - SNOBOL4, Perl

Character String Types

Evaluation
- Aid to writability
- As a primitive type with static length, they are inexpensive to provide -- why not have them?
- Dynamic length is nice, but is it worth the expense?

Implementation:
- Static length - compile-time descriptor
- Limited dynamic length - may need a run-time descriptor for length (but not in C and C++)
- Dynamic length - need run-time descriptor; allocation/deallocation is the biggest implementation problem

User-Defined Ordinal Types

- An *ordinal type* is one in which the range of possible values can be easily associated with the set of positive integers
- **Enumeration Types -**the user enumerates all of the possible values, which are given symbolic constants
- Can be used in For-loops, case statements, etc.
- Operations on ordinals include PRED, SUCC, ORD
- Usually cannot be I/O easily
- Mainly used for abstraction/readability

Examples

Pascal - cannot reuse constants; they can be used for array subscripts, for variables, case selectors; NO input or output; can be compared

Ada - constants can be reused (overloaded literals); disambiguate with context or type_name ‘ (one of them); can be used as in Pascal; can be input and output

C and C++ - like Pascal, except they can be input and output as integers

Java - does not include an enumeration type

Ada Example

- Some PLs allow a symbolic constant to appear in more than 1 type, Standard Pascal does not
- Ada is one of the few languages that allowed a symbol to name a value in more than one enumerated type.

```
Type letters is ('A', 'B', 'C', ...  'Z');
Type vowls is ('A', 'E', 'I', 'O', 'U');
```

- The following is ambiguous:

```
For letter in 'A' .. 'O' loop
```

- So Ada allows (requires) one to say:

```
For letter in vowels('A')..vowels('U') loop
```


Pascal Example

Pascal was one of the first widely used language to have good facilities for enumerated data types.

```
Type colorstype = (red, orange, yellow,
  green, blue, indigo, violet);
Var skinColor : colortype;
...
skinColor := blue;
...
If skinColor > green ...
...
For skinColor := red to violet do ...;
...
```


Subrange Type

- Limits a large type to a contiguous subsequence of values within the larger range, providing additional flexibility in programming and readability/abstraction
- Available in C/C++, Ada, Pascal, Modula-2
- Pascal Example
  Type upperCase ='A'..'Z'; lowerCase='a'..'z'; index =1..100;
- Ada Example
  - Subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants, e.g.

```
subtype POS_TYPE is  INTEGER range 0 ..INTEGER'LAST;
```


Ordinal Types Implementation

- Implementation is straightforward: enumeration types are implemented as non-negative integers
- Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Evaluation of Enumeration Types

- Aid to **efficiency** – e.g., compiler can select and use a compact efficient representation (e.g., small integers)
- Aid to **readability** -- e.g. no need to code a color as a number
- Aid to **maintainability** – e.g., adding a new color doesn't require updating hard-coded constants.
- Aid to **reliability** -- e.g. compiler can check operations and ranges of value.

Array Types

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- Design Issues include:
  - What types are legal for subscripts?
  - When are subscript ranges bound?
  - When does array allocation take place?
  - How many subscripts are allowed?
  - Can arrays be initialized at allocation time?
  - Are array slices allowed?

Array Indices

- An index maps into the array to find the specific element desired
  map(arrayName, indexValue) ? array element
- Usually placed inside of [ ] (Pascal, Modula-2, C, Java) or ( ) (FORTRAN, PL/I, Ada) marks
  - if the same marks are used for parameters then this weakens readability and can introduce ambiguity
- 2 types in an array definition
  - type of value being stored in array cells
  - type of index used
- Lower bound - implicit in C, Java and early FORTRAN

Subscript Bindings and Array Categories

Subscript Types:

FORTRAN, C - int only
Pascal - any ordinal type (int, boolean, char, enum)
Ada - int or enum (includes boolean and char)
Java - integer types only

Array Categories

Four Categories of Arrays based on subscript binding and binding to storage

1. Static - range of subscripts and storage bindings are static
- e.g. FORTRAN 77, some arrays in Ada
- *Advantage*: execution efficiency (no allocation or deallocation)

2. *Fixed stack dynamic* - range of subscripts is statically bound, but storage is bound at elaboration time.
- e.g. Pascal locals and C locals that are not static
- *Advantage*: space efficiency

Array Categories (continued)

3. *Stack-dynamic* - range and storage are dynamic, but fixed from then on for the variable's lifetime

e.g. Ada declare blocks

```
Declare
STUFF : array (1..N) of FLOAT;
  begin
  ...
  end;
```

Advantage: flexibility - size need not be known until the array is about to be used

Array Categories

4. *Heap-dynamic* - subscript range and storage bindings are dynamic and not fixed e.g. (FORTRAN 90)

INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT
(Declares MAT to be a dynamic 2-dim array)

ALLOCATE (MAT (10, NUMBER_OF_COLS))
(Allocates MAT to have 10 rows and NUMBER_OF_COLS columns)

DEALLOCATE MAT
(Deallocates MAT's storage)

- In APL & Perl, arrays grow and shrink as needed
- In Java, all arrays are objects (heap-dynamic)

Array dimensions

- Some languages limit the number of dimensions that an array can have
- FORTRAN I - limited to 3 dimensions
- FORTRAN IV and onward - up to 7 dimensions
- C/C++, Java - limited to 1 but arrays can be nested (i.e. array element is an array) allowing for any number of dimensions
- Most other languages have no restrictions

Array Initialization

- FORTRAN 77 - initialization at the time storage is allocated
```
INTEGER LIST(3)
Data list /0, 5, 5/
```
- C - length of array is implicit based on length of initialization list
```
int stuff [] = {2, 4, 6, 8};
Char name [] = ''Maryland'';
Char *names [] = {''maryland'', ''virginia'',
  delaware''};
```
- C/C++, Java - have optional initializations
- Ada - like C but you can specify which array elements are assigned values (instead of assigning all values)
```
SCORE : array (1..14,1..2) := (1=>(24,10), 2=>(10,7),
  3=>(12,30), others=>(0,0));
```
- Pascal, Modula-2 – don't have array initializations (Turbo Pascal does)

Array Operations

- Operations that apply to an array as a unit (as opposed to a single array element)
- Most languages have direct assignment of one array to another (A := B) if both arrays are equivalent
- FORTRAN: Allows array addition A+B
- Ada: Array concatenation A&B
- FORTRAN 90: library of Array ops including matrix multiplication, transpose
- APL: includes operations for vectors and matrices (transpose, reverse, etc...)

Array Operations in Java

- In Java, arrays are objects (sometimes called aggregate types)
- Declaration of an array may omit size as in:
  - int [ ] array1;
  - array1 is a pointer initialized to nil
  - at a later point, the array may get memory allocated it as with
  - array1 = new int [100];
- Array operations other than access (array1[2]) are through methods such as array1.length

Slices

A slice is some substructure of an array; nothing more than a referencing mechanism

1. FORTRAN 90 Example
   INTEGER MAT (1:4,1:4)
   INTEGER CUBE(1:4,1:4,1:4)
   MAT(1:4,1) - the first column of MAT
   MAT(2,1:4) - the second row of MAT
   CUBE(1:3,1:3,2:3) – 3x3x2 sub array
2. Ada Example
   single-dimensioned arrays only
   LIST(4..10)

Arrays

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Row major (by rows) or column major order (by columns)

An ***associative array*** is an unordered collection of data elements that are indexed by an equal number of values called *keys*

Design Issues:

1. What is the form of references to elements?
2. Is the size static or dynamic?

Perl's Associative Arrays

- Perl has a primitive datatype for hash tables aka "associative arrays".
- Elements indexed not by consecutive integers but by arbitrary keys
- %ages refers to an associative array and @people to a regular array
- Note the use of { }'s for associative arrays and [ ]'s for regular arrays

```
%ages = ("Bill Clinton"=>53,"Hillary"=>51, "Socks"=>"27 in cat years");
$ages{"Hillary"} = 52;b
@people=("Bill Clinton","Hillary","Socks");
$ages{"Bill Clinton"};        # Returns 53
$people[1];                   # returns "Hillary"
```

- keys(X), values (X) and each(X)

```
foreach $person (keys(%ages)) {print "I know the age of $person\n";}
foreach $age (values(%ages)){print "Somebody is $age\n";}
while (($person, $age) = each(%ages)) {print "$person is $age\n";}
```


Records

A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names

Design Issues:

1. What is the form of references?
2. What unit operations are defined?

Record Field References

- Record Definition Syntax -- COBOL uses level numbers to show nested records; others use familiar dot notation

  field_name OF rec_name_1 OF ... OF rec_name_n
  rec_name_1.rec_name_2.....rec_name_n.field_name
- *Fully qualified references* must include all record names
- *Elliptical references* allow leaving out record names as long as the reference is unambiguous
- With clause in Pascal and Modula2

```
With employee.address do
   begin
     street := '422 North Charles St.';
     city := 'Baltimore';
     zip := 21250
   end;
```

Record Operations

1.Assignment
- Pascal, Ada, and C allow it if the types are identical
  – In Ada, the RHS can be an aggregate constant

2. Initialization
- Allowed in Ada, using an aggregate constant

3. Comparison
- In Ada, = and /=; one operand can be an aggregate constant

4. MOVE CORRESPONDING (Cobol)
   Move all fields in the source record to fields with the same names in the destination record

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


Records and Arrays

Comparing records and arrays

1. Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
2. Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Union Types

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?

3. Is a variant tag or discriminant required?

Examples: Unions

1.FORTRAN - with EQUIVALENCE

2. Algol 68 - discriminated unions
- Use a hidden tag to maintain the current type
- Tag is implicitly set by assignment
- References are legal only in conformity clause

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

- This runtime type selection is a safe method of accessing union objects

Pascal Union Types

Problem with Pascal's design: type checking is ineffective. Reasons:

User can create inconsistent unions (because the tag can be individually assigned)

```
var blurb : intreal;
x : real;
blurb.tag := true;   { it is an integer }
blurb.blint := 47;   { ok }
blurb.tag := false;  { it is a real }
x := blurb.blreal;   { assigns an integer to a real }
```

The tag is optional! Now, only the declaration and the second and last assignments are required to cause trouble

Pascal Union Types

Pascal has record variants which support both discriminated & nondiscriminated unions, e.g.

```
type shape = (circle, triangle, rectangle);
    colors = (red,green,blue);
    figure = record
            filled: boolean;
            color: colors;
            case form: shape of
              circle: (diameter: real);
              triangle: (leftside: integer; rightside: integer; angle:real);
              rectangle: (side1: integer; side2: integer)
              end;
```


Pascal Union Types

```
case myfigure.form of
 circle : writeln('It is a circle; its diameter is', myfigure.diameter);
 triangle : begin
   writeln('It is a triangle');
   writeln('its sides are:' myfigure.leftside, myfigure.rightside);
   wtiteln('the angle between the sides is :', myfigure.angle);
   end;
 rectangle : begin
   writeln('It is a rectangle');
   writeln('its sides are: 'myfigure.side1, myfigure.side2)
   end
end
```


Pascal Union Types

But, Pascal allowed for problems because:

- The user could explicitly set the record variant tag
  `myfigure.form := triangle`
- The variant tag is option. We could have defined a figure as:

```
Type figure = record ...
        case shape of
          circle: (diameter: real);
          ...
        end
```

Pascal's variant records introduce potential type problems, but are also a loophole which allows you to do, for example pointer arithmetic.

Ada Union Types

Ada only has “discriminated unions”

These are safer than union types in Pascal & Modula2 because:

- The tag must be present
- It is impossible for the user to create an inconsistent union (because tag cannot be assigned by itself -- All assignments to the union must include the tag value)

Union Types

C and C++ have only free unions (no tags)

- Not part of their records
- No type checking of references

6. Java has neither records nor unions

Evaluation - potentially unsafe in most languages (not Ada)

Set Types

- A *set* is a type whose variables can store unordered collections of distinct values from some ordinal type
- *Design Issue:*
  - What is the maximum number of elements in any set base type?
- Usually implemented as a bit vector.
  - Allows for very efficient implementation of basic set operations (e.g., membership check, intersection, union)

Sets in Pascal

- No maximum size in the language definition and implementation dependant and usually a function of hardware word size (e.g., 64, 96, …).
- Result: Code not portable, poor writability if max is too small
- Set operations: union (+), intersection (*), difference (-), =, <>, superset (>=), subset (<=), in

```
Type colors = (red,blue,green,yellow,orange,white,black);
    colorset = set of colors;
    var s1, s2 : colorset;
    …
    s1 := [red,blue,yellow,white];
    s2 := [black,blue];
```

Examples

2. Modula-2 and Modula-3
- Additional operations: INCL, EXCL, / (symmetric set difference (elements in one but not both operands))

3. Ada - does not include sets, but defines in as set membership operator for all enumeration types

4. Java includes a class for set operations

Evaluation

- If a language does not have sets, they must be simulated, either with enumerated types or with arrays

- Arrays are more flexible than sets, but have much slower operations

Implementation

- Usually stored as bit strings and use logical operations for the set operations.

Pointers

A *pointer type* is a type in which the range of values consists of memory addresses and a special value, nil (or null)

Uses:

1. Addressing flexibility
2. Dynamic storage management

Design Issues:

- What is the scope and lifetime of pointer variables?
- What is the lifetime of heap-dynamic variables?
- Are pointers restricted to pointing at a particular type?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should a language support pointer types, reference types, or both?

Fundamental Pointer Operations

- Assignment of an address to a pointer
- References (explicit versus implicit dereferencing)

Problems with pointers

1. Dangling pointers (dangerous)
   - A pointer points to a heap-dynamic variable that has been deallocated
   - Creating one:
     - Allocate a heap-dynamic variable and set a pointer to point at it
     - Set a second pointer to the value of the first pointer
     - Deallocate the heap-dynamic variable, using the first pointer

Problems with pointers

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
     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*

Problems with Pointers

1. *Pascal:* used for dynamic storage management only
   - Explicit dereferencing
   - Dangling pointers are possible (dispose)
   - Dangling objects are also possible
2. *Ada:* a little better than Pascal and Modula-2
   - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's scope
   - All pointers are initialized to null
   - Similar dangling object problem (but rarely happens)

Pointer Problems: C and C++

- Used for dynamic storage management and addressing
- Explicit dereferencing and address-of operator
- Can do address arithmetic in restricted forms
- Domain type need not be fixed (void *)

```
float stuff[100];
float *p;
p = stuff;
```

*(p+5) is equivalent to stuff[5] and p[5]
*(p+i) is equivalent to stuff[i] and p[i]
void * - can point to any type and can be type checked (cannot be dereferenced)

Pointer Problems: Fortran 90

- Can point to heap and non-heap variables
- Implicit dereferencing
- Special assignment operator for non dereferenced references

REAL, POINTER :: ptr *(POINTER is an attribute)*
ptr => target *(where target is either a pointer or a non-pointer with the TARGET attribute)*
The TARGET attribute is assigned in the declaration, e.g.
INTEGER, TARGET :: NODE

Pointers

5. C++ Reference Types
   - Constant pointers that are implicitly dereferenced
   - Used for parameters
   - Advantages of both pass-by-reference and pass-by-value
6. Java - Only references
   - 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

Memory Management

- Memory management: identify unused, dynamically allocated memory cells and return them to the heap
- Approaches
  - Manual: explicit allocation and deallocation (C, C++)
  - Automatic:
    - » Reference counters (modula2, Adobe Photoshop)
    - » Garbage collection (Lisp, Java)
- Problems with manual approach:
  - Requires programmer effort
  - Programmer's failures leads to space leaks and dangling references/sharing
  - Proper explicit memory management is difficult and has been estimagte to account for up to 40% of development time!

Reference Counting

- Idea: keep track how many references there are to a cell in memory. If this number drops to 0, the cell is garbage.
- Store garbage in free list; allocate from this list
- Advantages
  - immediacy
  - resources can be freed directly
  - immediate reuse of memory possible
- Disadvantages
  - Can't handle cyclic data structures
  - Bad locality properties
  - Large overhead for pointer manipulation

Garbage Collection (GC)

- GC is a process by which dynamically allocated storage is reclaimed during the execution of a program.
- Usually refers to automatic periodic storage reclamation by the garbage collector (part of the run-time system), as opposed to explicit code to free specific blocks of memory.
- Usually triggered during memory allocation when available free memory falls below a threshold. Normal execution is suspended and GC is run.
- Major GC algorithms:
  - Mark and sweep
  - Copying
  - Incremental garbage collection algorithms

Mark and Sweep

- Oldest and simplest algorithm
- Has two phases: mark and sweep
- Collection algorithms: When program runs out of memory, stop program, do garbage collection and resume program.
- Here: Keep free memory in *free pool*. When allocation encounters empty free pool, do garbage collection.
- **Mark**: Go through live memory and mark all live cells.
- **Sweep**: Go through whole memory and put a reference to all non-live cells into free pool.

Evaluation of pointers

- Dangling pointers and dangling objects are problems, as is heap management
- Pointers are like goto's -- they widen the range of cells that can be accessed by a variable
- Pointers are necessary--so we can't design a language without them

Summary

This chapter covered Data Types, a large part of what determines a language's style and use. It discusses primitive data types, user defined enumerations and sub-range types. Design issues of arrays, records, unions, set and pointers are discussed along with reference to modern languages.