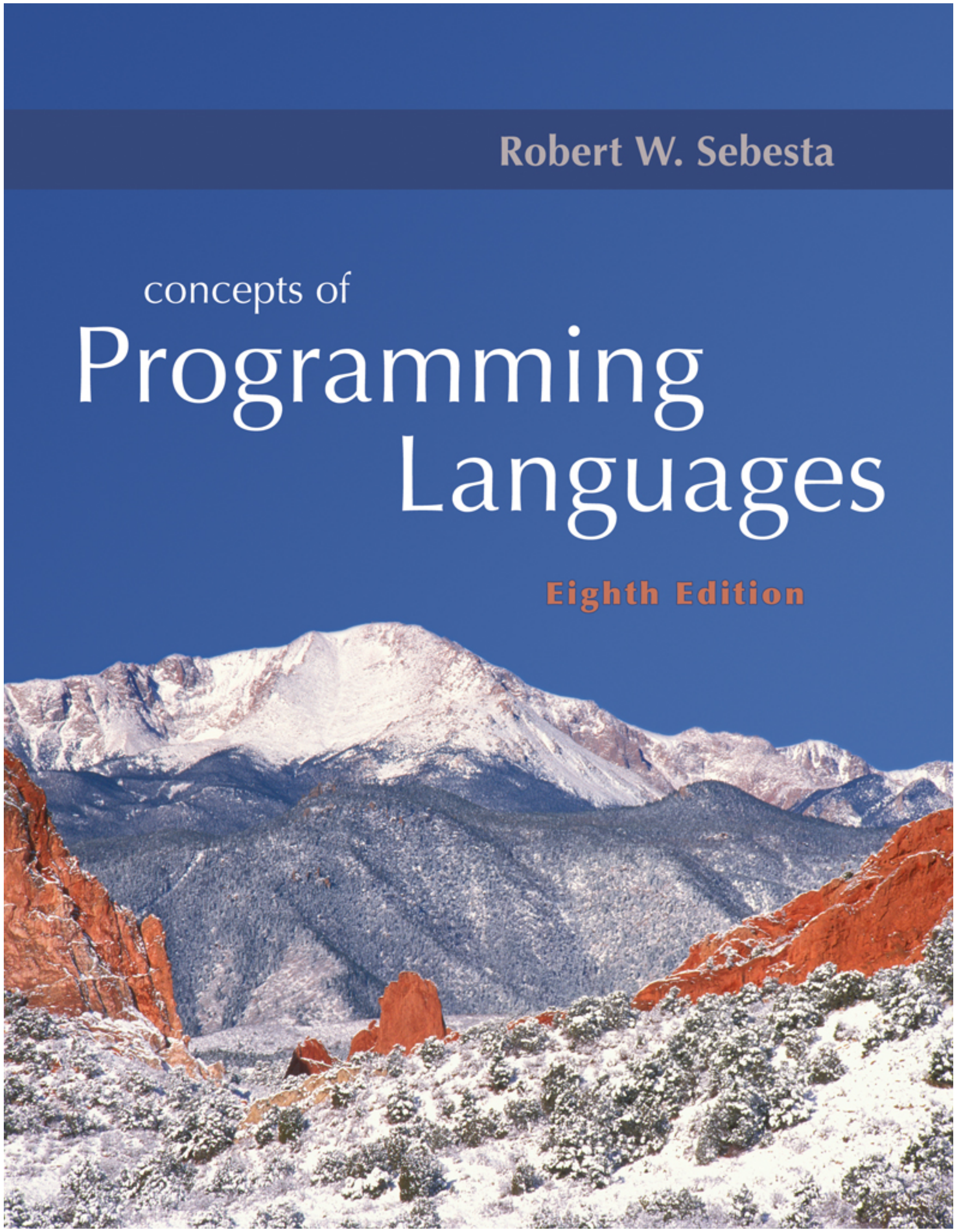
Chapter 6

Data Types
part 2
(updated to 11th edition)



ISBN 0-321-49362-1

Record Types

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition

```
– 01 EMP-REC.
–    02 EMP-NAME.
–       05 FIRST  PIC X(20).
–       05 MID    PIC X(10).
–       05 LAST   PIC X(20).
–    02 HOURLY-RATE  PIC 99V99.
```

Definition of Records in Ada

- Record structures are indicated in an orthogonal way (nested example)

```
type Emp_Name_Type is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
end record;
type Emp_Rec_Type is record
        Emp_Name: Emp_Name_Type;
        Hourly_Rate: Float;
end record;
```

Definition of Records in C++

- Nested example (more similar to Ada)

```
struct Emp_Name_Type {
  string first;
  string middle;
  string last;
};

struct Emp_Rec_Type {
  Emp_Name_Type Emp_name;
  float hourly_rate;
}
```

References to Records

- Record field references
  - 1. COBOL
  - field_name OF record_name_1 OF ... OF record_name_n
  - 2. Others (dot notation)
  - record_name_1.record_name_2. ... record_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, for example in COBOL
- FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Operations on Records

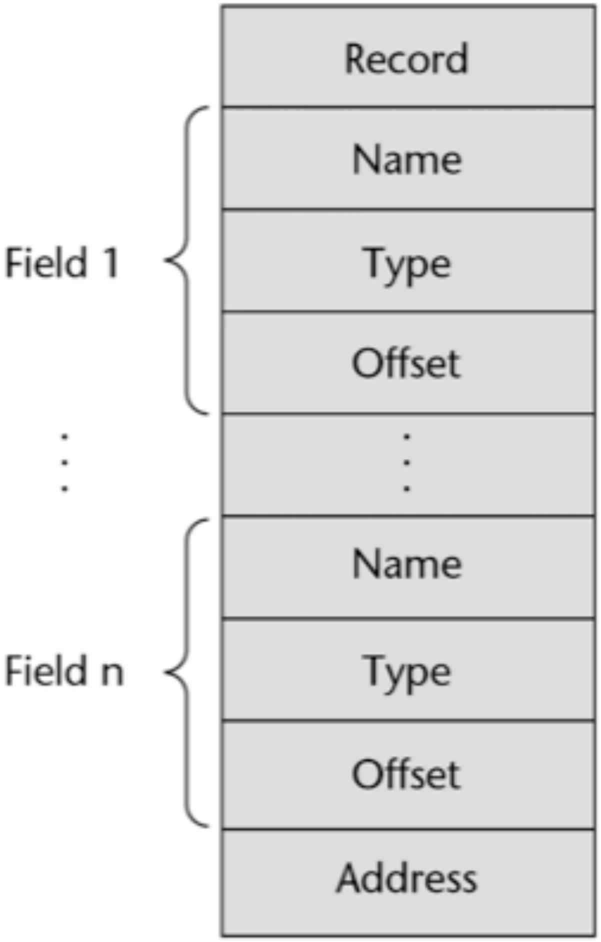
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
  - Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Unions Types

- A union is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

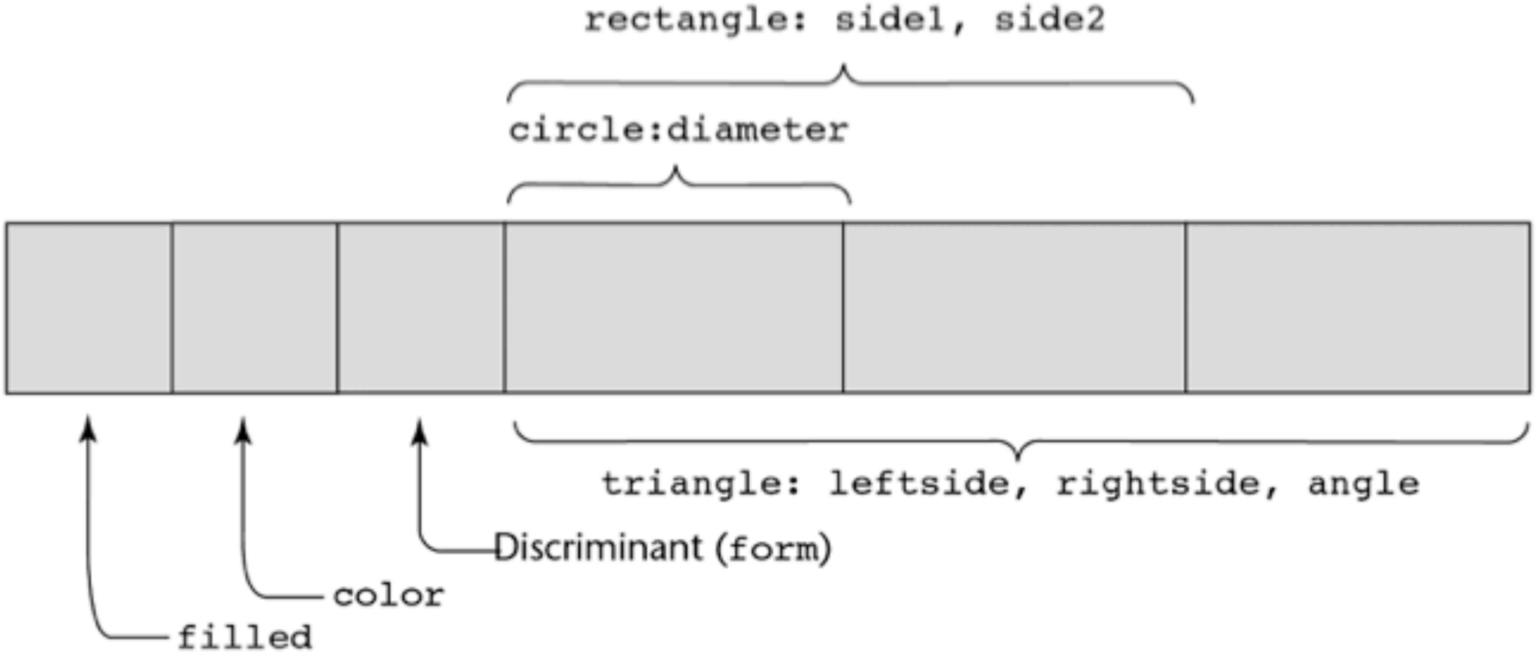
Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union
- Type checking of unions require that each union include a type indicator called a discriminant
  - Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

Ada Union Type Illustrated



- A discriminated union of three shape variables

Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- Ada's discriminated unions are safe

Pointer and Reference Types

- A pointer type variable has a range of values that consists of memory addresses and a special value, nil
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap)

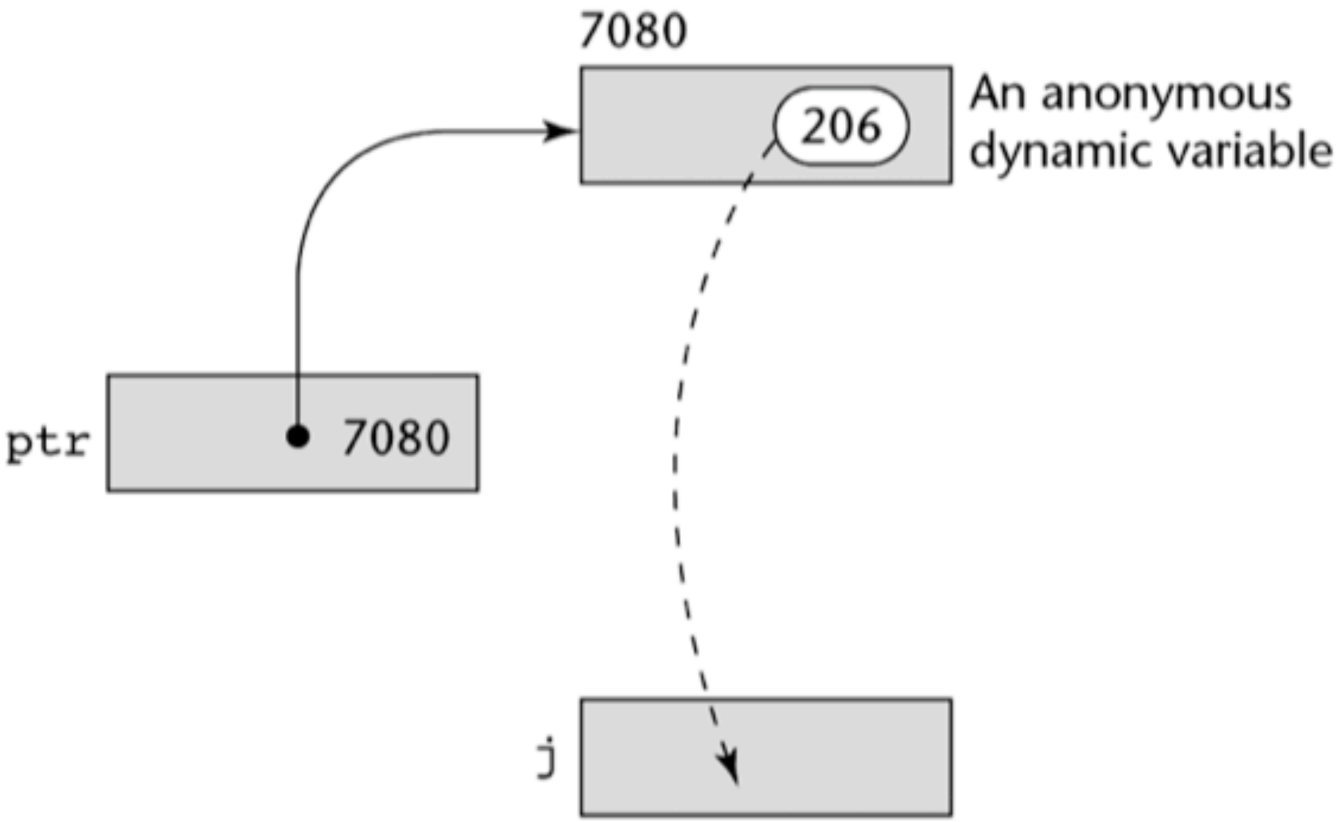
Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *
  - j = *ptr
  - sets j to the value located at ptr

Pointer Assignment Illustrated



- The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called garbage)
    - Pointer p1 is set to point to a newly created heap-dynamic variable
    - Pointer p1 is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called memory leakage

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada (possible with UNCHECKED_DEALLOCATION)

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators

Pointer Arithmetic in C and C++

- float list[100];
- float *p;
- p = list;

- *(p+5) is equivalent to list[5] and p[5]
- *(p+i) is equivalent to list[i] and p[i]

- Domain type need not be fixed (**void** *)
-     void * can point to any type and can be type
-       checked (cannot be de-referenced)

Reference Types

- C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++, must include 'unsafe' modifier
- Smalltalk, Python, Ruby, Lua: all variables are references; always implicitly dereferenced

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 or references are necessary for dynamic data structures--so we can't design a language without them

Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

Dangling Pointer Problem

- Tombstone: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space – no popular languages use this..
- Locks-and-keys: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer. Used in UW-Pascal (compiler of Pascal)
- Best solution: out of hands of programmer (implicit deallocation: Java; C# references)

Heap Management

- One of design goals of LISP was that reclamation of unused cells not task of programmer (most LISP data consists of cells in linked list)
- A very complex run-time process
- Single-size cells vs. variable-size cells
- Fundamental design question: When should deallocation be performed?

Heap Management

- Fundamental design question: When should deallocation be performed?
- Two approaches to reclaim garbage
  - Reference counters (eager): reclamation is gradual
  - Mark-sweep (lazy approach): reclamation occurs when the list of variable space becomes empty

Reference Counter

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - Disadvantages: space required, execution time required to change counters, complications for cells connected circularly
  - Advantage: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

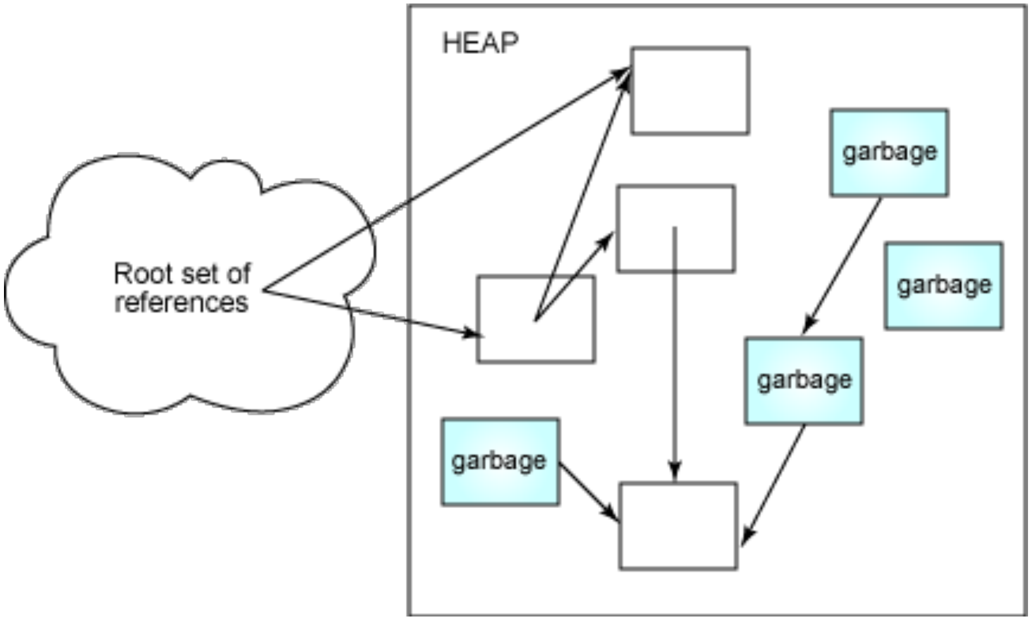
- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins to gather garbage

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins to gather garbage
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution.
    Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm

Cartoon from https://www.ibm.com/developerworks/library/j-jtp10283/



Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages, since cells store values of variables of any type
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult (one solution: each cell has cell size as first field)
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management