Topics

- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types

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.
```

References to Records

- Record field references

  1. COBOL

  field_name OF record_name_n OF ... OF record_name_1

  2. Others (dot notation)

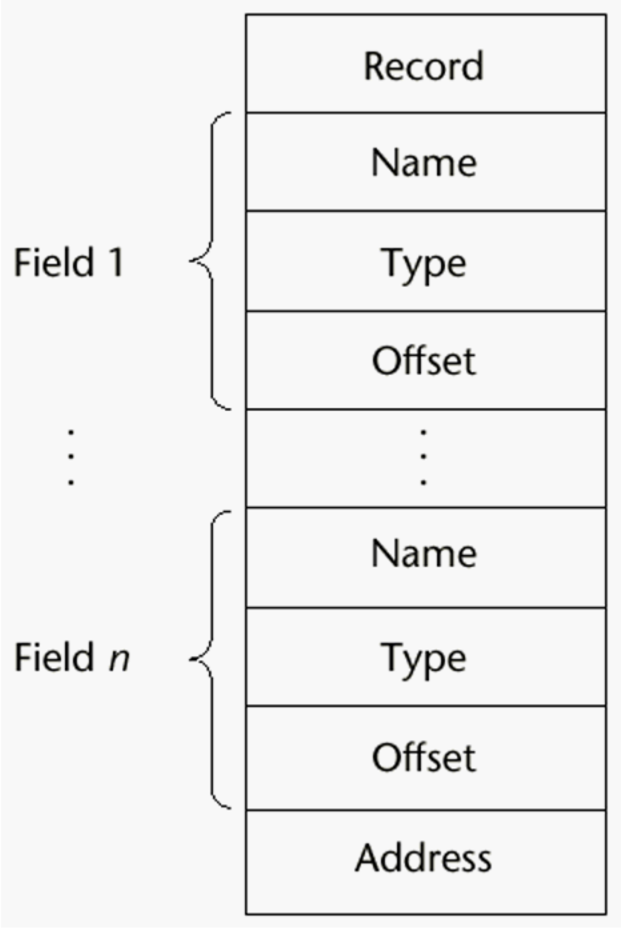
  record_name_1.record_name_2. ... record_name_n.field_name

- 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

Implementation of Record Type

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



Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to its lists, but immutable
    - Create with a tuple literal

      ```
      myTuple = (3, 5.8, 'apple')
      ```

    - Referenced with subscripts
    - Catenation with `+` and deleted completely with **`del`**

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

  - Access as follows:

    `#1(myTuple)` is the first element

  - A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
let a, b, c = tup
```

  This assigns a tuple to a tuple pattern `(a, b, c)`

List Types

- Lists in Lisp and Scheme are delimited by parentheses and use no commas

  `(A B C D)` and `(A (B C) D)`

- Data and code have the same form

  As data, `(A B C)` is literally what it is

  As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`

- The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

  `'(A B C)` is data

List Types (continued)

- List Operations in Scheme
  - `CAR` returns the first element of its list parameter

    `(CAR '(A B C))` returns `A`
  - `CDR` returns the remainder of its list parameter after the first element has been removed

    `(CDR '(A B C))` returns `(B C)`
  - `CONS` puts its first parameter into its second parameter, a list, to make a new list

    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` returns a new list of its parameters

    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

List Types (continued)

- List Operations in ML
  - Lists are written in brackets and the elements are separated by commas
  - List elements must be of the same type
  - The Scheme `CONS` function is a binary operator in ML, `::`

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
  - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class
- Python Lists
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common Lisp, ML, and F#, Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]`   Sets `x` to `5.8`
  - List elements can be deleted with `del`

    `del myList[1]`
  - List Comprehensions – derived from set notation

    ```
    [x * x for x in range(6) if x % 3 == 0]
    ```

    `range(6)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell's List Comprehensions
  - The original

```
[n * n | n <- [1..10]]
```

- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> (i * i) |]
```

- Both C# and Java support lists through their generic heap-dynamic collection classes, `List` and `ArrayList`, respectively

Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issue
  - Should type checking be required?

Discriminated vs. Free Unions

- 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/tag*
  - Supported by ML, Haskell, and F#

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

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 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?

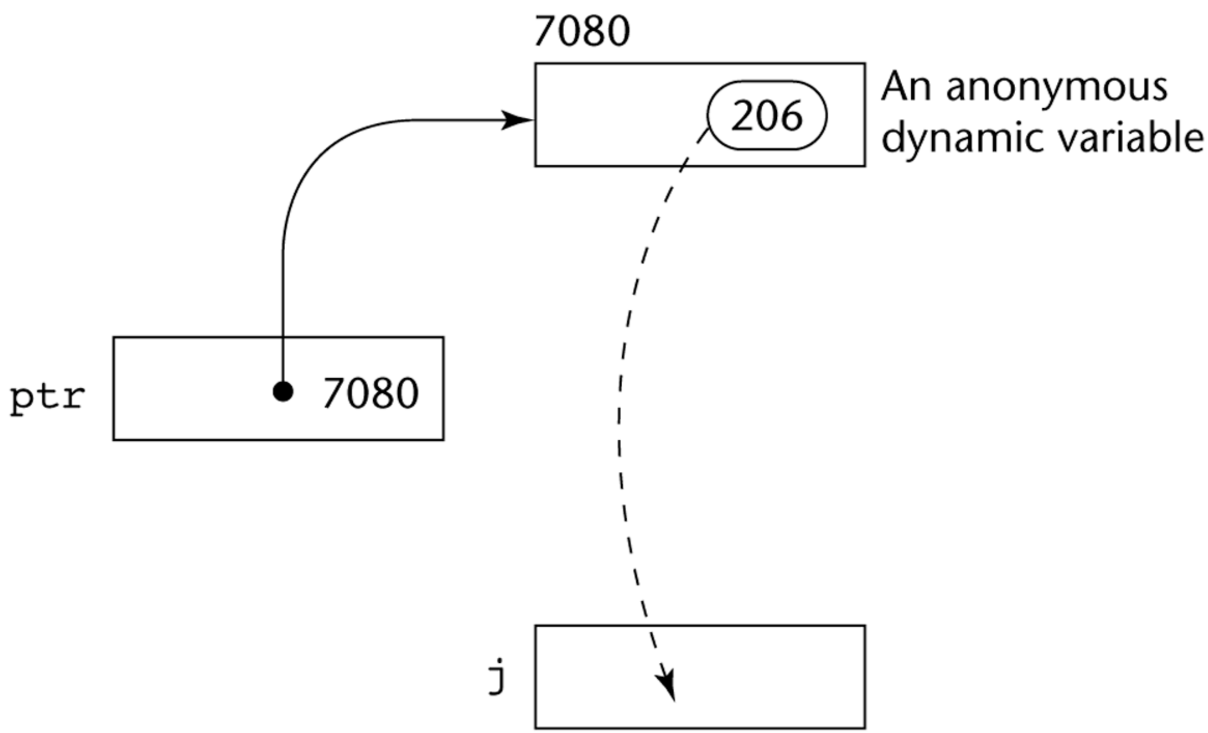
Pointer Operations

- 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 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
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (**`void *`**)

  `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
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]`

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++

Evaluation of Pointers

- Dangling pointers and dangling objects are problems
- 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

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
- *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

Heap Management

- A very complex run-time process
- Single-size cells vs. variable-size cells
- Two approaches to reclaim garbage
  - Reference counters (*eager approach*): 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, 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
  - 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

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

Assignments

- Reading assignment: Chapter 6
- Written assignment: assignment two (4%), due on February 8