Basic Types and Arrays

- **Basic Types**
  - integer, real (floating point), boolean (0,1), character

- **Arrays**
  - A[0..99] : integer array
    
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 99 |
    |---|---|---|---|---|---|---|---|---|-----|
    |   |   |   |   |   |   |   |   |   |     |
    | A[5] |

Records and Pointers

- **Record (also called a struct)**
  - Group data together that are related
    - X : complex pointer
      - real_part : real
      - imaginary_part : real
  - To access the fields we use “dot” notation.
    - X.real_part
    - X.imaginary_part

Record Definition

- **Record definition creates a new type**
  - Definition
    - record complex : (r
      - real_part : real,
      - imaginary_part : real
    )
  - Use in a declaration
    - X : complex
**Pointer**

- A pointer is a reference to a variable or record (or object in Java world).

  ![Diagram of a pointer and blob]

- In C, if X is of type pointer to Y then *X is of type Y

**Creating a Record**

- We use the “new” operator to create a record.

  ```
  P : pointer to blob;
  P := new blob;
  ```

**Simple Linked List**

- A linked list
  - Group data together in a flexible, dynamic way.
  - We’ll describe several list ADTs later.

  ![Diagram of a linked list]

**Application**

**Sparse Polynomials**

- \[ 10 + 4x^2 + 20x^{40} + 8x^{86} \]

  ```
  record poly : {
    exp : integer,
    coef : integer,
    next : poly pointer
  }
  ```

  ![Diagram of a sparse polynomial]
**Identically Zero Polynomial**

P null pointer

P 1 2 86

**Addition of Polynomials**

10 + 4x^2 + 20x^40 + 8x^86

Q 7x + 10x^2 - 8x^86

**Recursive Addition**

Add(P, Q : poly pointer): poly pointer{
R : poly pointer
case {
P = null : R := Q ;
Q = null : R := P ;
P.exp < Q.exp : R := P ;
    R.next := Add(P.next,Q);
P.exp > Q.exp : R := Q ;
    R.next := Add(P,Q.next);
P.exp = Q.exp : R := P ;
    R.coef := P.coef + Q.coef ;
    R.next := Add(P.next,Q.next);
}
return R
}

**Example**

Add

P 10 0 2 40 86

Q 7 10 2 86 -8
Example (first call)

During the Recursive Call

The Recursive Call

After the Recursive Call
Notes on Addition

- Addition is destructive, that is, the original polynomials are gone after the operation.
- We don’t salvage “garbage” nodes. Let’s talk about this.
- We don’t consider the case when the coefficients cancel. Let’s talk about this.

Unneeded nodes to Garbage

- How would you force the unneeded node to be garbage in the code on slide 11?
- Suggestions?

Memory Management – Global Allocator

- Global Allocator’s store – always get and return blocks to global allocator – an area in the memory from which we can dynamically allocate memory.
  - The user (the program) must ‘free’ the memory when done.
Memory Management – Garbage Collection

- Garbage collection – run time system recovers inaccessible blocks from time-to-time. Used in Lisp, Smalltalk, Java.
  - No need to return blocks to an allocator.
  - Care must be taken to make unneeded blocks inaccessible.
  - When garbage collection kicks in there may be undesirable response time.

Solution for Polyn. Addition

```plaintext
P.exp = Q.exp : R := P;
    R.coef := P.coef + Q.coef;
    if R.coef = 0 then
        R := Add(P.next,Q.next);
    // The terms with coef = 0 have been removed from the result
    else
        R.next := Add(P.next,Q.next);
    Free(P); Free(Q);
```

Use of Global Allocator

```plaintext
P.exp = Q.exp : R := P;
    R.coef := P.coef + Q.coef;
    if R.coef = 0 then
        R := Add(P.next,Q.next);
        Free(P); Free(Q);
    else
        R.next := Add(P.next,Q.next);
        Free(Q);
```