Pointers (review and examples)

CSE 373
Data Structures
Lecture 2
Basic Types and Arrays

• Basic Types
  › integer, real (floating point), boolean (0,1), character

• Arrays
  › A[0..99] : integer array

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array}
\]

Records and Pointers

• **Record** (also called a *struct*)
  › Group data together that are related
    
    ```
    X : complex pointer
    |
    v
    real_part : real
    |
    v
    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

```plaintext
record complex : (  
  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).

\[ X : \text{blob pointer} \]

\[ *X \rightarrow \text{blob} \]

- 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.

\[ P : \text{pointer to blob;} \]

\[ P \quad \text{(null pointer)} \]

\[ P := \text{new blob;} \]
Simple Linked List

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

$L : \text{node pointer}$

```
record node : (
    data : integer,
    next : node pointer
)
```
Application
Sparse Polynomials

• $10 + 4x^2 + 20x^{40} + 8x^{86}$

record poly : (  
    exp : integer,  
    coef : integer,  
    next : poly pointer  
)
Identically Zero Polynomial

P null pointer

\[
\begin{array}{ccc}
& 1 & 2 & 86 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
Addition of Polynomials

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

\[ P \]

\[ 0 \quad 2 \quad 40 \quad 86 \]

\[ 10 \quad 4 \quad 20 \quad 8 \]

\[ 7x + 10x^2 - 8x^{86} \]

\[ Q \]

\[ 1 \quad 2 \quad 86 \]

\[ 7 \quad 10 \quad -8 \]
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

Q
Example (first call)
The Recursive Call

Add

```
<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>86</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>86</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-8</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

P -> R -> Q
During the Recursive Call

Add

Return value

Represent return values

R

10

1

7

2

14

2

10

86

-8

0

20

40

86

0

12/26/03 Pointers - Lecture 2
After the Recursive Call
The final picture

R

0
10

2
14

40
20

86
0

1
7

2
86
10

2
86
-8

unneeded
garbage
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 – Private Store

- **Private store** – get blocks from a private store when possible and return them when done.
  - Efficiently uses blocks of a specific size
  - The list of unused blocks can build up eventually using too much memory.
Private Store

\[ \begin{array}{c}
\text{unneeded} \\
\text{garbage}
\end{array} \]

\[ \begin{array}{c}
R \\
0 \\
10 \\
1 \\
7 \\
2 \\
14 \\
2 \\
10 \\
2 \\
86 \\
-8
\end{array} \]
Private Store

```
\[ R \quad 0 \quad 10 \quad 1 \quad 7 \quad 2 \quad 14 \quad 40 \quad 20 \]
```

```
\text{FreeList}
R \quad 86 \quad 0 \quad 86 \quad -8 \quad 2 \quad 10
```

12/26/03 Pointers - Lecture 2 22
Memory Management – Global Allocator

• **Global Allocator’s store** – always get and return blocks to global allocator
  + Necessary for dynamic memory.
  + Blocks of various sizes can be merged if they reside in contiguous memory.
  - Allocator may not handle blocks of different sizes well.
  - Allocator may be slower than a private store.
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 or keep them in a private store.
  - Care must be taken to make unneeded blocks inaccessible.
  - When garbage collection kicks in there may be undesirable response time.
Solution for Polyn. Addition

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);
     }

12/26/03
Use of Private Store or Global Allocator

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);
    
}