CS 126 Lecture P3: Data Structures

Outline

(This is a hard lecture--study these slides after class.)

- **Introduction**
- Array
- Structure
- Linked list
- Implementation: C pointer

Why Data Structures?

Users' views: students, bank records, ...
????
C basics: int, float, char, ...
Memory elements

- Users' needs
  - What to do when we have a large amount of data to deal with?
  - Want to organize it in ways that are easy-to-understand
  - Want to be space-efficient
  - Want to be time-efficient
- What hardware gives us
  - Just a bunch of uniform, individually addressable storage elements
- Want to bridge the gap between the abstractions

Data Type and Data Structure

Data TYPE

* set of possible values for variables
* operations on those values

Ex: int, float, char, ...

Data STRUCTURE

* collection of related values
* mechanism for organizing information

Examples [stay tuned]:

- built-in: array, struct
- linked: linked list, binary tree
- compound: array of structs, list of trees

- READ SEDGEWICK, SECTIONS 3.1, 3.2, 3.3

Outline

- ~~Introduction~~
- **Array**
- Structure
- Linked list
- Implementation: C pointer

Array

Memory address	Index	Content
100	0	**h**
101	1	**e**
102	2	**l**
103	3	**l**
104	4	**o**

array name: **`word`**
3rd letter: **`word[2]`**

(analogy: seats and students)

Fundamental data structure

- HOMOGENEOUS collection of values
  (all the same type)
- store values sequentially in memory
- associate INDEX with each value
- use array name and index
  to quickly access kth for any k

Array (cont.)

Concise and efficient method for working with large collections of data values

- Most important limitation:
  need to know size ahead of time

- Natural applications
  - vector, matrix
  - spreadsheet
  - string of characters

  ...

Computer memory is a huge array
(array abstraction is easily implemented)

Example of array use

Symbolic manipulation of polynomials

C representation of x^9 + 3x^5 + 7:

```
int a[10];
for (i = 0; i < 10; i++)
   a[i] = 0;
a[0] = 7; a[5] = 3; a[9] = 1;
```

x^9+3x^5+7

Possible memory representation of x^9 + 3x^5 + 7

0	100: 7	7	$[x^0]$
1	101: 0		
2	102: 0		
3	103: 0		
4	104: 0		
5	105: 3	3	$[x^5]$
6	106: 0		
7	107: 0		
8	108: 0		
9	109: 1	1	$[x^9]$

Use exponents as array indices

Store coefficients in the array

Memory address

(assumes array stored in locations 100-109)

Advantages of array use for this application:
   can get to each item quickly
   index carries implicit info, takes no space
Disadvantage: Uses up space for unused items

Histogram program sample run

Histogram of grades for a recent COS126 class

```
% more histo.data
68 69 67 65 60 68 67 94 59 54 21 96 95 94 55 60
64 65 93 54 64 94 63 65 89 93 28 92 62 61 87 92
89 51 95 85 88 94 86 93 93 84 86 93 84 92 93 92
90 84 87 88 90 85 87 91 87 87 82 85 85 88 87 80
90 85 81 85 84 78 90 85 79 91 92 75 78 89 76 93
75 78 89 76 90 81 74 78 80 76 80 81 84 76 78 74
79 84 74 77 84 78 76 82 70 77 83 76 74 70 83 70
72 73 67 81 72 69 84 83 71 72 84 73 82 83 70 72
69 82 82 73 80

% cc histo.c

% a.out < histo.data
10
20 **
30
40
50 ******
60 *****************
70 **********************************
80 ******************************************
90 ***************************
```


Sample program: compute histogram

Based on key feature of array:
use data as index

HISTOGRAM: bar graph of number of occurences
of each data value

Ex: class grades in 0-99 range
how many in 0-9, 10-19. 20-29, 30-39, ... ranges

```
#include <stdio.h>
main()
  { int i, j, val;
    int h[10];
    for (i = 0; i < 10; i++) h[i] = 0;
    while (scanf("%d", &val) != EOF)
      h[val/10]++;
    for (j = 0; j < 10; j++)
      {
        printf("%2d-%2d ", j*10, j*10+9);
        for (i = 0; i < h[j]; i++)
          printf("*");
        printf(" ");
      }
```

Initialize all the histo-bins to 0.

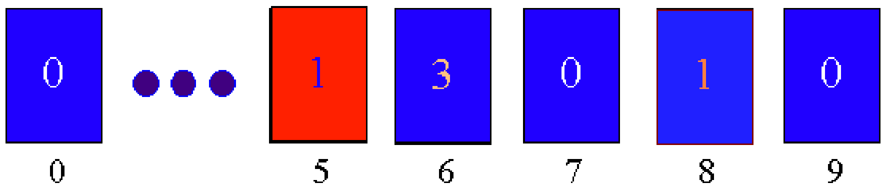
```
{i = val / 10;
h[i]++;  }
```

Calculate which bin;
Increment that bin;

for all bins
print right # of stars for each

See also Program 3.7 in Sedgewick

Demo 1



Outline

- ~~Introduction~~
- ~~Array~~
- **Structure**
- Linked list
- Implementation: C pointer

Structures

Fundamental data structure
- HETEROGENEOUS collection of values (possibly different types)
- store values in FIELDS
- associate NAME with each field
- use struct name and field name to access value

(analogy: bag of potentially different things)

- Built-in C mechanism: struct

Basis for building "user-defined types"

- Applications
  - database records
  - linked list nodes (stay tuned)
  - ...

Ex: C representation of C students

```
struct student
   { char name [20]; float grade; };
struct student t, x, y;
x.name = "Bill Gates"; x.grade = 60.0;
y.name = "Steve Jobs"; y.grade = 70.0;
...
if (x.grade > y.grade) t = x; else t = y;
printf("Better student: %s ", t.name);
```


typedef

User definition of type names

- Main use: put type descriptions in one place (makes code more portable)

Ex:

```
typedef float Grade;
typedef char name [20] Name;
struct student
   { Name name; Grade grade; };
struct student t, x, y;
```

- Common use: avoid typing "struct" (makes code more concise)

Ex:

```
struct student
   { char name [20]; float grade; };
typedef struct student Student;
Student t, x, y;
```

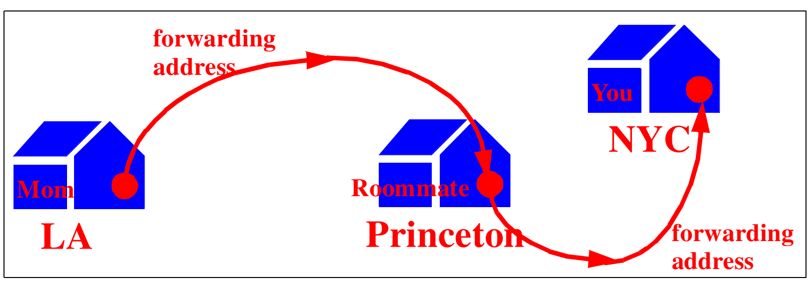
Ex:

```
typedef struct { int p; int q; } Rational;
float x; Rational t;
x = (1.0)*t.p/t.q
```

Outline

- ~~Introduction~~
- ~~Array~~
- ~~Structure~~
- **Linked list**
- Implementation: C pointer

Linked List

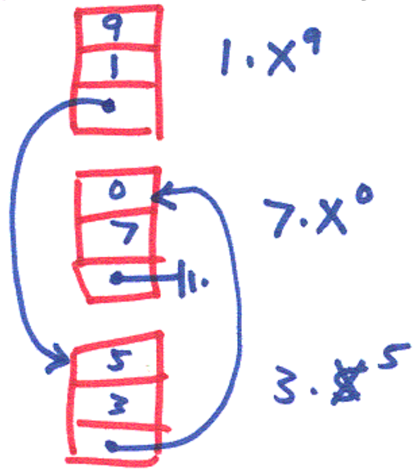


Fundamental data structure
- homogeneous collection of values
  (all the same type)
- store values ANYWHERE in memory
- associate LINK with each value
- use link to quickly access the NEXT value

- "Dynamic allocation": allocate houses on demand

- Possible memory representation of x^9 + 3x^5 + 7

```
100: 9
101: 1
102: 240
...
198: 0
199: 7
200: 000
...
240: 5
241: 3
242: 198
```



Advantage: space proportional to amount of info
Disadvantage: can only get to *next* item quickly

- C specification of x^9 + 3x^5 + 7: ???

Need to know:
  - how to associate pieces of information
  - how to specify links
  - how to reserve memory to be used
  - how to use links to access information

Linked vs. Sequential allocation

- Polynomial example illustrates tradeoffs
  SPARSE polynomial: few terms, large exponent
  ex: x^1000000 + 5 x^50000 + 12
  DENSE polynomial: few nonzero coefficients
  ex: x^7 + x^6 + 3x^4 + 2x^3 + 1

- Huge sparse polynomial

	array	linked list
SPACE	huge	tiny
TIME		
count terms	huge	tiny
find coeffient of x^k	instant	tiny

- Huge dense polynomial

	array	linked list
SPACE	huge	3*huge
TIME		
count terms	huge	huge
find coeffient of x^k	instant	huge

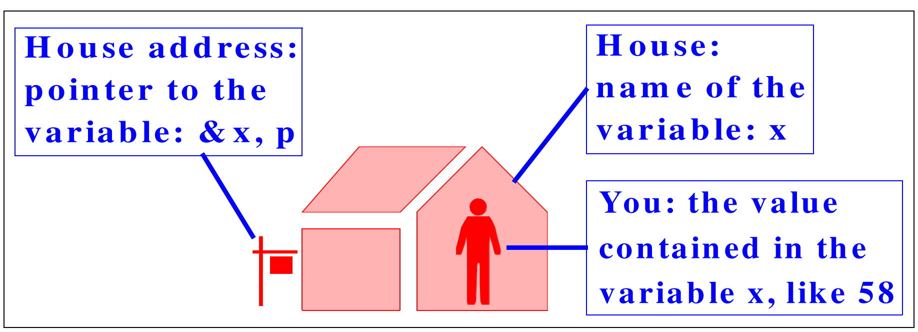
Digression: a few programming axioms
* know space and time costs
* there is never enough time or space
* it is easy to write programs that waste both
* you will not notice until it matters

- More examples of linked vs. sequential:
  Programs 3.5 and 3.9 in Sedgewick

Outline

- ~~Introduction~~
- ~~Array~~
- ~~Structure~~
- ~~Linked list~~
- **Implementation: C pointer**
  - pointers and simple variables
  - pointers and arrays
  - pointers and linked lists
  - for each of these, understand how to
    + declare the variables involved
    + how to initialize them
    + how to use them

Pointer

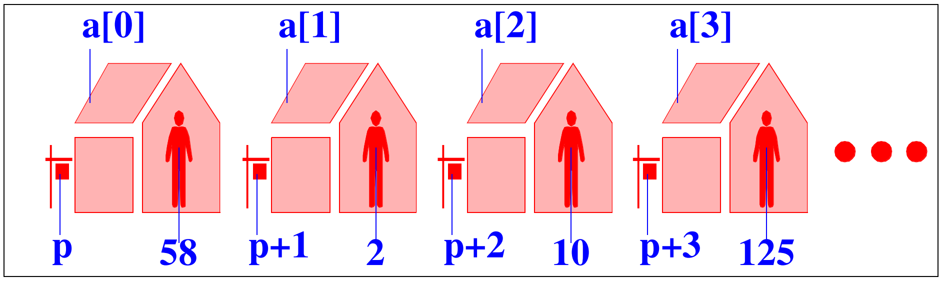


`int x;`	build a house of type int and name x
`int *p;`	p can contain an address to any int-type house (decl)
`p = &x;`	p is now the address of house x (init)
`x = 58;`	the person 58 moves into house x
`*p = 58;`	the person 58 moves into the house at address p (use)

- `&x` and `p` are equivalent (`&` returns address of house)
- `x` and `*p` are equivalent (`*` gets to house at address)

Pointer and Array

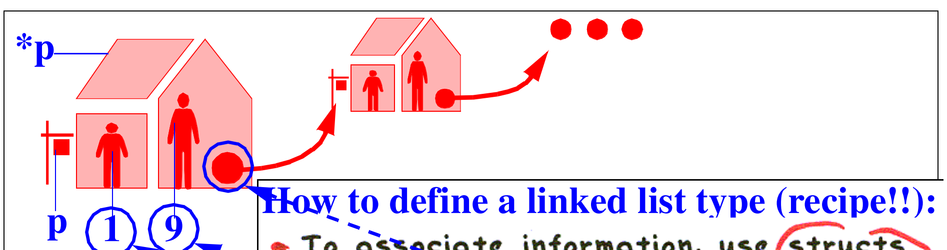
```
int a[100]; int *p; p = &a[0]; *p = 58;...
```



- p — pointer to array (first item)
- *p — first array item
- p+1 — pointer to second array item
- *(p+1) — second array item
- *(p+i) — (i+1)st array item
- p[i] — shorthand for the same thing

- `&x[i]` and `p+i` are equivalent
- `x[i]` and `*(p+i)` are equivalent

Pointer and Linked List



How to define a linked list type (recipe!!):

- To associate information, use structs
- To specify links, use "*" (pointers)

```
typedef struct node* link;
struct node
   { int coef; int exp; link next; };
link p;
```

The meaning of variables and fields:

- p — pointer to structure
- *p — structure
- (*p).coef — field in structure
- p->coef — shorthand for the same thing

but I haven't even told you how to initialize the pointers yet!!

- To reserve memory for a structure, use "malloc"

```
p = malloc(sizeof *p);
```

  malloc is a library function in stdlib.h
  sizeof gives the number of memory words
  needed for a node (*p is a node)
  malloc reserves that much memory somewhere
  and returns a pointer to it

- To use a pointer to access information, use "->"

```
p->coef = 1;
```

Ex! Build the list for x^9 + 3x^5 + 7

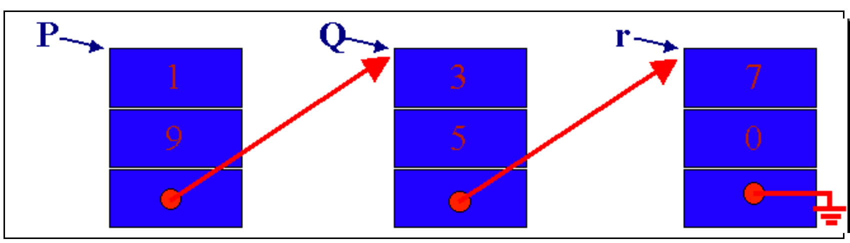
```
p = malloc(sizeof *p)
p->coef = 1; p->exp = 9;
q = malloc(sizeof *p)
q->coef = 3; q->exp = 5;
r = malloc(sizeof *p)
r->coef = 7; r->exp = 0;
p->next = q; q->next = r; r->next = NULL;
```

Much more next lecture!

[NULL is a special "no link" indicator]

STUDY THIS CODE: Tip of the iceberg!

Demo 2



Closing

- Whew!
- Lots of material in this lecture.
- Pointers are confusing.
- Study these lecture slides.