Lecture P8: Pointers and Linked Lists

Lewis Caroll
Through the Looking Glass

“"The name of the song is called 'Haddocks' Eyes.'""

“"Oh, that's the name of the song, is it?" Alice said, trying to feel
interested.

“"No, you don't understand," the Knight said, looking a little vexed.
"That's what the name is called. The name really is 'The Aged
Man.'""

“"Then I ought to have said 'That's what the song is called'?" Alice
corrected herself.

“"No, you oughtn't: that's quite another thing! The song is called
'Ways and Means,' but that is only what it's called, you know!"

“"Well, what is the song, then?" said Alice, who was by this time
completely bewildered.

"I was coming to that," the Knight said. "The song really is 'A-sitting
On A Gate,' and the tune's my own invention."

Lewis Caroll
Through the Looking Glass

Lecture P8: Pointers and Linked Lists

Pointers Overview

Basic computer memory abstraction.
- Indexed sequence of bits.
- Address = index.
- Ex 1: TOY.
  - basic unit = word = 16 bits
  - 8-bit address refers to one of 256 words
- Ex 2: Arizona.
  - basic unit = byte = 8 bits
  - 32-bit address refers to one of 4 billion+ bytes

Pointer = VARIABLE that holds memory address.
- Allow function to change inputs.
- Create self-referential data structures.
- Better understanding of arrays.

Pointers in TOY

Variable that stores the value of a single MEMORY ADDRESS.
- In TOY, memory addresses are 00 – FF.
  - indirect addressing: store a memory address in a register
  - Very powerful and useful programming mechanism.
    - more confusing in C than in TOY
    - easy to abuse!

<table>
<thead>
<tr>
<th>Address</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>9</td>
<td>E5</td>
<td>.</td>
<td>7</td>
<td>0</td>
<td>00</td>
<td>.</td>
</tr>
</tbody>
</table>

Memory location D2 stores a "pointer" to another memory location (E5) of interest.

Pointers Intuition

Pointer abstraction captures distinction between a thing and its name.

<table>
<thead>
<tr>
<th>Thing</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web page</td>
<td><a href="http://www.princeton.edu">www.princeton.edu</a></td>
</tr>
<tr>
<td>Email inbox</td>
<td><a href="mailto:august@cs.princeton.edu">august@cs.princeton.edu</a></td>
</tr>
<tr>
<td>This room</td>
<td>Frist 302</td>
</tr>
<tr>
<td>Bank account</td>
<td>45-234-23310076</td>
</tr>
<tr>
<td>Princeton student</td>
<td>610080478</td>
</tr>
<tr>
<td>Word of TOY memory</td>
<td>1A</td>
</tr>
<tr>
<td>Byte of PC memory</td>
<td>FFBEBFB2</td>
</tr>
<tr>
<td>int x;</td>
<td>&amp;x</td>
</tr>
<tr>
<td>*px</td>
<td>px declared as int *px;</td>
</tr>
</tbody>
</table>
Pointers in C

C pointers.
- If \( x \) is an integer:
  \&\( x \) is a pointer to \( x \) (memory address of \( x \))
- If \( px \) is a pointer to an integer:
  \*\( px \) is the integer

```c
#include <stdio.h>
int main(void) {
    int x;
    int *px;
    x = 7;
    px = &x;
    printf(" x = %d\n", x);
    printf(" px = %p\n", px);
    printf("*px = %d\n", *px);
    return 0;
}
```

Allocate storage for pointer to int

Pointers as Arguments to Functions

Goal: function that swaps values of two integers.

A first attempt:

```c
#include <stdio.h>
void swap(int a, int b) {
    int t;
    t = a; a = b; b = t;
}
int main(void) {
    int x = 7, y = 10;
    swap(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

### badswap.c

changes value stored in memory address for \( x \) and \( y \)

Pointers as Arguments to Functions

Now, one that works.

```c
#include <stdio.h>
void swap(int *pa, int *pb) {
    int t;
    t = *pa; *pa = *pb; *pb = t;
}
int main(void) {
    int x = 7, y = 10;
    swap(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```

Linked List Overview

Goal: deal with large amounts of data.
- Organize data so that it is easy to manipulate.
- Time and space efficient.

Basic computer memory abstraction.
- Indexed sequence of bits (words, bytes).
- Address \( = \) index.

Need higher level abstractions to bridge gap.
- Array.
- Struct.
- LINKED LIST
- Binary tree.
- Database.
- . . .
**Linked List**

Fundamental data structure.
- HOMOGENEOUS collection of values (all same type).
- Store values ANYWHERE in memory.
- Associate LINK with each value.
- Use link for immediate access to the NEXT value.

Possible TOY memory representation of $x^9 + 3x^5 + 7$.
- Assume linked list starts in location D0.

<table>
<thead>
<tr>
<th>Address</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>...</th>
<th>D9</th>
<th>DA</th>
<th>DB</th>
<th>...</th>
<th>E5</th>
<th>E6</th>
<th>E7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>9</td>
<td>E5</td>
<td>..</td>
<td>7</td>
<td>0</td>
<td>00</td>
<td>..</td>
<td>3</td>
<td>5</td>
<td>D9</td>
</tr>
</tbody>
</table>

---

**Linked List**

Fundamental data structure.
- HOMOGENEOUS collection of values (all same type).
- Store values ANYWHERE in memory.
- Associate LINK with each value.
- Use link for immediate access to the NEXT value.

Possible TOY memory representation of $x^9 + 3x^5 + 7$.
- Assume linked list starts in location D0.

<table>
<thead>
<tr>
<th>Address</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>...</th>
<th>D9</th>
<th>DA</th>
<th>DB</th>
<th>...</th>
<th>E5</th>
<th>E6</th>
<th>E7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>9</td>
<td>E5</td>
<td>..</td>
<td>7</td>
<td>0</td>
<td>00</td>
<td>..</td>
<td>3</td>
<td>5</td>
<td>D9</td>
</tr>
</tbody>
</table>

---

**Linked List vs. Array**

Polynomial example illustrates basic tradeoffs.
- Sparse polynomial = few terms, large exponent.
  Ex. $3x^{1000000} + 5x^{20000} + 7$

<table>
<thead>
<tr>
<th>array</th>
</tr>
</thead>
</table>

| linked list |

- Dense polynomial = mostly nonzero coefficients.
  Ex. $6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + 1$

| array |

| linked list |
### Linked List vs. Array

Polynomial example illustrates basic tradeoffs.

<table>
<thead>
<tr>
<th></th>
<th>Huge Sparse Polynomial</th>
<th>Huge Dense Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>array</strong></td>
<td>huge</td>
<td>3 * huge</td>
</tr>
<tr>
<td><strong>linked</strong></td>
<td>tiny</td>
<td>huge</td>
</tr>
<tr>
<td><strong>space</strong></td>
<td>huge</td>
<td>tiny</td>
</tr>
<tr>
<td><strong>time</strong></td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

**Lesson:** know space and time costs.
- Axiom 1: there is never enough space.
- Axiom 2: there is never enough time.

Time to determine coefficient of \( x^4 \):

### Overview of Linked Lists in C

Not directly built into C language. Need to know:

**How to associate pieces of information.**
- User-define type using `struct`.
- Include `struct` field for coefficient and exponent.

**How to specify links.**
- Include `struct` field for POINTER to next linked list element.

**How to reserve memory to be used.**
- Allocate memory DYNAMICALLY (as you need it).
- `malloc()`

**How to use links to access information.**
- `->` and `. ` operators

### Linked List for Polynomial

**C code to represent** \( x^9 + 3x^5 + 7 \).

- **Statically**, using nodes.
- **Dynamically**, using links.

```c
struct node {
    int coef;
    int exp;
    struct node *next;
};

#define N 3

int main(void) {
    struct node p, q, r;
    p.coef = 1; p.exp = 9;
    q.coef = 3; q.exp = 5;
    r.coef = 7; r.exp = 0;
    p.next = &q;
    q.next = &r;
    r.next = NULL;
    return 0;
}
```

**poly1.c**

### Linked List for Polynomial

**C code to represent** \( x^9 + 3x^5 + 7 \).

- **Statically**, using nodes.
- **Dynamically**, using links.

```c
#include <stdlib.h>
typedef struct node *link;
struct node {
    int coef;
    int exp;
    struct node *next;
};

int main(void) {
    link x, y, z;
    x = malloc(sizeof *x);
    x->coef = 1; x->exp = 9;
    y = malloc(sizeof *y);
    y->coef = 3; y->exp = 5;
    z = malloc(sizeof *z);
    z->coef = 7; z->exp = 0;
    x->next = y;
    y->next = z;
    z->next = NULL;
    return 0;
}
```

**poly2.c**
Review of Stack ADT

Create ADT for stack.
- Lecture P5: implement using an array.
- Now: re-implement using linked list.

client uses data type, without regard to how it is represented or implemented.

- include "STACK.h"
- void STACKinit(void);
- int STACKisempty(void);
- void STACKpush(int item);
- int STACKpop(void);
- void STACKshow(void);

Stack Implementation With Linked Lists

- include <stdlib.h>
- include "STACK.h"
- typedef struct STACKnode* link;
- struct STACKnode {
  int item;
  link next;
};
- head
- void STACKinit(void) {
  head = NULL;
}
- int STACKisempty(void) {
  return head == NULL;
}
- void STACKpush(int item) {
  head = NEWnode(item, head);
}
- int STACKpop(void) {
  int value; link second;
  if (head == NULL) {
    printf("Stack underflow.\n");
    exit(EXIT_FAILURE);
  }
  value = head->item; second = head->next;
  free(head);
  head = second;
  return value;
}
- void STACKshow(void) {
  link x;
  for (x = head; x != NULL; x = x->next)
    printf("%d\n", x->item);
}
Implementing Stacks: Arrays vs. Linked Lists

We can implement a stack with either array or linked list, and switch implementation without changing interface or client.

- gcc client.c stacklist.c
- gcc client.c stackarray.c

Which is better for stacks?
- Array
  - Requires upper bound MAX on stack size.
  - Uses space proportional to MAX.
- Linked List
  - No need to know stack size ahead of time.
  - Space proportional to number of elements.
  - Requires extra space to store pointers.
  - Dynamic memory allocation is slower.

Conclusions

Whew, lots of material in this lecture!
- Pointers are useful, but can be confusing. Bewildering, even.
- Study these slides and carefully read relevant material.
- Do not debug by speculatively sprinkling &’s and *’s in your program!
- Instead, do draw pictures with boxes and arrows.
  - Alice should have done this!

Lecture P8: Extra Slides

Pointers and Arrays

avg.c

```
#include <stdio.h>
#define N 64

int main(void) {
    int a[N] = {84, 67, 24, ..., 89, 90};
    int i, sum;

    for (i = 0; i < N; i++)
        sum += a[i];

    printf("%d\n", sum / N);
    return 0;
}
```

"Pointer arithmetic"

- &a[0] = a+0 = D000
- &a[1] = a+1 = D004
- &a[2] = a+2 = D008
- a[0] = *a = 84
- a[1] = *(a+1) = 67
- a[2] = *(a+2) = 24

<table>
<thead>
<tr>
<th>Memory address</th>
<th>D000</th>
<th>D004</th>
<th>D008</th>
<th>..</th>
<th>D0F8</th>
<th>D0FC</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>84</td>
<td>67</td>
<td>24</td>
<td>..</td>
<td>89</td>
<td>90</td>
<td>..</td>
</tr>
</tbody>
</table>
Pointers and Arrays

"Pointer arithmetic"

\[2[a] = *(2+a) = 24\]

This is legal C, but don’t ever do this at home!!!

<table>
<thead>
<tr>
<th>Memory address</th>
<th>D000</th>
<th>D004</th>
<th>D008</th>
<th>..</th>
<th>D0F8</th>
<th>D0FC</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>84</td>
<td>67</td>
<td>24</td>
<td>..</td>
<td>89</td>
<td>90</td>
<td>..</td>
</tr>
</tbody>
</table>

Passing Arrays to Functions

Pass array to function.
- Pointer to array element 0 is passed instead.

```c
#include <stdio.h>
#define N 64

int average(int b[], int n) {
    int i, sum;
    for (i = 0; i < n; i++)
        sum += b[i];
    return sum / n;
}

int main(void) {
    int a[N] = {84, 67, 24, .., 89, 90};
    printf("%d
", average(a, N));
    return 0;
}
```

Why Pass Array as Pointer?

Advantages.
- Efficiency for large arrays – don’t want to copy entire array.
- Easy to pass "array slice" of "sub-array" to functions.

```c
int average(int b[], int n) {
    int i, sum;
    for (i = 0; i < n; i++)
        sum += b[i];
    return sum / n;
}
```

Passing Arrays to Functions

Many C programmers use \texttt{int *b} instead of \texttt{int b[]} in function prototype.
- Emphasizes that array decays to pointer when passed to function.

```c
int average(int *b, int n) {
    int i, sum;
    for (i = 0; i < n; i++)
        sum += b[i];
    return sum / n;
}
```