Lecture P8: Pointers and Linked Lists

Pointer Overview

Basic computer memory abstraction.
- Indexed sequence of bits.
- Address = index.

Pointer = variable that stores memory address.

Uses.
- Allow function to change inputs.
- Better understanding of arrays.
- Create "linked lists."

Pointers

Variable that stores the value of a single MEMORY LOCATION.
- In TOY, memory locations are 00 – FF.
  - Indexed addressing: store a memory location in a register
- Very powerful and useful programming mechanism.
- Confusing and easy to abuse!

<table>
<thead>
<tr>
<th>Address</th>
<th>D000</th>
<th>D004</th>
<th>D008</th>
<th>. .</th>
<th>D0C8</th>
<th>D0CC</th>
<th>D0D0</th>
<th>. .</th>
<th>D200</th>
<th>D204</th>
<th>D208</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>9</td>
<td>1</td>
<td>D200</td>
<td>. .</td>
<td>0</td>
<td>7</td>
<td>0000</td>
<td>. .</td>
<td>5</td>
<td>3</td>
<td>D0C8</td>
</tr>
</tbody>
</table>

Memory location D008 stores a "pointer" to another memory location of interest.

Pointers in C

C pointers.
- If x is an integer
  &x is a pointer to x.
- If px is a pointer to an integer
  *px is the integer.

Unix

% gcc pointer.c
% a.out
x = 7
px = ffbefb24
*px = 7

#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;
}
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

only swaps copies of x and y

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;
}
```

swap.c

changes value stored in memory address for x and y

Pointers and Arrays

```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;
}
```

avg.c

"Pointer arithmetic"

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0] = a = 84</td>
<td>a[1] = *(a+1) = 67</td>
<td>a[2] = *(a+2) = 24</td>
</tr>
</tbody>
</table>

Value

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

Passing Arrays to Functions

In C, when array is passed to a function, a pointer to first element of array is passed.

```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\n", average(a, N));
    return 0;
}
```

avg.c

receive the value D000 from main

passes &a[0] = D000 to function
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;
}
```

```c
int main(void) {
    . . .
    res = average(a+5, 10);
    . . .
}
```

Passing Arrays to Functions

Some C programmers use `int *b` instead of `int b[]` in function prototype to emphasize that arrays decay to pointers when passed to functions.

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

an equivalent function

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

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.
- 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 memory representation of \( x^9 + 3x^5 + 7 \).
- Assume linked list starts in location D000.

<table>
<thead>
<tr>
<th>Address</th>
<th>D000</th>
<th>D004</th>
<th>D008</th>
<th>D0CB</th>
<th>DOCC</th>
<th>D0D0</th>
<th>. .</th>
<th>D200</th>
<th>D204</th>
<th>D208</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0000</td>
<td>.</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

special "NULL" memory address denotes end of list
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 memory representation of $x^9 + 3x^5 + 7$.
- Assume linked list starts in location D000.

```
<table>
<thead>
<tr>
<th>Address</th>
<th>D000</th>
<th>D004</th>
<th>D008</th>
<th>D0CB</th>
<th>D0CC</th>
<th>D0D0</th>
<th>D200</th>
<th>D204</th>
<th>D208</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>9</td>
<td>1</td>
<td>D200</td>
<td>0</td>
<td>7</td>
<td>0000</td>
<td>5</td>
<td>3</td>
<td>D0C8</td>
</tr>
</tbody>
</table>
```

Head — 9 1 D200 0 7 0000 NULL 5 3 D0C8

Linked List vs. Array

Polynomial example illustrates basic tradeoffs.
- Sparse polynomial = few terms, large exponent.
  Ex. $x^{1000000} + 5x^{50000} + 7$
- Dense polynomial = mostly nonzero coefficients.
  Ex. $x^7 + x^6 + 3x^4 + 2x^3 + 1$

### Space vs. Time Tradeoffs

Axiom 1: there is never enough space.
Axiom 2: there is never enough time.

It is easy to write programs that waste both.
- You will not notice until it matters.

Lesson: know space and time costs.

### Overview of Linked Lists in C

Not directly built in to 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 of \( x^9 + 3x^5 + 7 \).
- Statically, using nodes.

---

**poly1.c**

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

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;
}

---

**poly2.c**

#include <stdlib.h>

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

int main(void) {
    link x, y, z;
    x = malloc(sizeof(struct node));
    x->coef = 1; x->exp = 9;
    y = malloc(sizeof(*link));
    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;
}

---

**poly3.c**

#include <stdlib.h>

link NEWnode(int c, int e, link n) {
    link x = malloc(sizeof *x);
    if (x == NULL) {
        printf("Out of memory.\n");
        exit(EXIT_FAILURE);
    }
    x->coef = c; x->exp = e; x->next = n;
    return x;
}

int main(void) {
    link x, y, z;
    x = NEWnode(1, 9, NULL);
    y = NEWnode(3, 5, NULL);
    z = NEWnode(7, 0, NULL);
    x->next = y;
    y->next = z;
    z->next = NULL;
    return 0;
}

---

Better Programming Style

Write separate function to handle memory allocation and initialization.

---

**poly3.c**

Check if malloc fails

---

Review of Stack Interface

In Lecture P5, we created ADT for stack.
- We implemented stack using arrays.
- Now, we give alternate implementation using linked lists.

---

**STACK.h**

#include "STACK.h"

void STACKInit(void);
int STACKisempty(void);
void STACKpush(int);
int STACKpop(void);

---

**client.c**

int main(void) {
    int a, b;
    . . .
    STACKInit();
    STACKpush(a);
    . . .
    b = STACKpop();
    return 0;
}
Stack Implementation With Linked Lists

```c
#include <stdlib.h>
#include <assert.h>
#include "STACK.h"

typedef struct STACKnode* link;
struct STACKnode {
    int item;
    link next;
};

static link list; void STACKinit(void) {
    list = NULL;
}

int STACKisempty(void) {
    return NULL == list;
}

link NEWnode(int item, link next) {
    link x = malloc(sizeof *x);
    if (x == NULL) {
        printf("Out of memory.\n");
        exit(EXIT_FAILURE);
    }
    x->item = item; x->next = next;
    return x;
}

void STACKpush(int item) {
    link x = NEWnode(item, list);
    head = x;
}

int STACKpop(void) {
    int item = list->item;
    link x = list->next;
    free(list);
    list = x;
    return 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
```

OR
```
%gcc client.c stackarray.c
```

Which is better?

- Array
- Linked List
Conclusions

Whew, lots of material in this lecture!

Pointers are useful, but confusing.

Study these slides and carefully read relevant material.