Pointers and Linked Lists

Pointer Overview
- Indexed sequence of binary numbers.
- 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>D00C</th>
<th>D010</th>
<th>D014</th>
<th>D018</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>
</tr>
</tbody>
</table>

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

Pointer Intuition
- Mailbox corresponds to unit of computer memory.
- Postal address corresponds to memory address (pointer).
Pointers in C

- If `x` is an integer
  - `&x` is the memory location of `x`.
- If `px` is a pointer to an integer
  - `*px` is the integer.

```c
#include <stdio.h>

int main(void) {
    int x = 7;
    int *px = &x;
    printf("x = %d\n", x);
    printf("px = %p\n", px);
    printf("*px = %d\n", *px);
    return 0;
}
```

Unix

```
% gcc pointer.c
% a.out
x = 7
px = ffefeb24
*px = 7
```

Pointers as Arguments to Functions

Goal: write a function to swap the values of two different integer variables.

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;
}
```

```c
badswap.c

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;
}
```

Pointers as Arguments to Functions

Goal: write a function to swap the values of two different integer variables.

```
#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

#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;
}
```

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;
}
```

```c
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;
}
```

```
Memory address: D000 D004 D008 .. D0F8 D0FC ..
```

<table>
<thead>
<tr>
<th>Value</th>
<th>84</th>
<th>67</th>
<th>24</th>
<th>..</th>
<th>89</th>
<th>90</th>
<th>..</th>
</tr>
</thead>
</table>

"Pointer arithmetic"

<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0] = a+0 = D000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1] = a+1 = D004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2] = a+2 = D008</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Why Pass Array as Pointer?

- 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;
}
```

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 binary numbers.
- Address = index.

Need higher level abstractions to bridge gap.
- Array
- Struct
- LINKED 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>D0C8</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>.</td>
<td>.</td>
<td>0</td>
<td>7</td>
<td>0000</td>
<td>.</td>
</tr>
</tbody>
</table>

- exponent
- coefficient
- memory address of next element

**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. $x^7 + x^6 + 3x^4 + 2x^3 + 1$

**Huge Sparse Polynomial**
- array: huge
- linked: tiny

**Huge Dense Polynomial**
- array: huge
- linked: $3 *$ huge

**Time to determine coefficient of $x^k$.**
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 into C language. Need to know:

**How to associate pieces of information.**
  * User-define type using `struct`.

**How to specify links.**
  * Linked list element contains information (coefficient and exponent) and MEMORY LOCATION of next linked list element.
  * Need to use pointers.

**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.
- Dynamically using links.

**poly1.c**

```c
typedef struct node *link;
struct node {
  int coef;
  int exp;
  link next;
};
define node to store two integers

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**

```c
#include <stdlib.h>
typedef struct node *link;
struct node {
  . . .
};
define node to store two integers

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;
}
```

---

Linked List for Polynomial

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

- Statically, using nodes.
- Dynamically using links.

**Study this code: tip of iceberg!**

- Allocate enough memory to store node
- Link up nodes of list
- Initialize data
- Link up nodes of list
- Allocate enough memory to store node
- Study this code: tip of iceberg!
**Better Programming Style**

Write separate function to handle memory allocation and initialization.

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

link NEWnode(int c, int e, link n) {
    link x = malloc(sizeof(struct node));
    assert(x != NULL);
    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;
}
```

---

**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.

```c
#include "STACK.h"

void STACKinit(void);
int  STACKempty(void);
void STACKpush(int);
int  STACKpop(void);
```

```c
int main(void) {
    int a, b;
    . . .
    STACKinit();
    STACKpush(a);
    . . .
    b = STACKpop();
    return 0;
}
```

---

**Stack Implementation With Linked Lists**

Push and pop at the front of list.

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

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

static link head;

void STACKinit(void) {
    head = NULL;
}

int STACKempty(void) {
    return NULL == head;
}

link NEWnode(int item, link next) {
    link x = malloc(sizeof *x);
    assert(x != NULL);
    x->item = item; x->next = next;
    return x;
}

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

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

---

**Stack Implementation With Linked Lists (cont)**

Allocate memory and initialize new node

```c
link NEWnode(int item, link next) {
    link x = malloc(sizeof *x);
    assert(x != NULL);
    x->item = item; x->next = next;
    return x;
}

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

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

Switch implementation without changing interface or client.
%gcc client.c stacklist.c
OR
%gcc client.c stackarray.c

Implementing Stacks: Arrays vs. Linked Lists

We can implement a stack with either array or linked list. Which is better?

Array
- Requires upper bound MAX on stack size.
- Uses space proportional to MAX.

Linked List
- No need to know stack size ahead of time.
- Requires extra space to store pointers.
- Dynamically allocating memory with malloc slows down code.

Conclusions

Whew, lots of material in this lecture!

Pointers are useful, but confusing.

Study these slides and carefully read relevant material.