Lecture P8: Pointers and Linked Lists

Pointers 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 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.
- Confusing and easy to abuse!

<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>

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

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

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

pointer.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;
}
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;
}
```

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;
}
```

changes value stored in memory address for x and y

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.
- ...

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>256GB</td>
<td>1</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>

head
1 9 E5
7 0 00
NULL
3 5 D9
256GB 1
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. $x^{1000000} + 5x^{500000} + 7$
- Dense polynomial = mostly nonzero coefficients.
  Ex. $x^7 + x^6 + 3x^4 + 2x^3 + 1$

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

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

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.

```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;
}
```
**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 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;
}
```

**Better Programming Style**

Better Programming Style

```c
#include <stdlib.h>

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

int main(void) {
  link x = NULL;
  x = NEWnode(7, 0, x);
  x = NEWnode(3, 5, x);
  x = NEWnode(1, 9, x);
  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"

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

**Stack Implementation With Linked Lists**

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

typedef struct STACKnode* link;
struct STACKnode {
  int item;
  link next;
};
static link head = NULL;

void STACKinit(void) {
  head = NULL;
}

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

void STACKpush(int item) {
  int item;
  link next;
};

void STACKpop(void) {
  static link head = NULL;
};

int STACKisempty(void) {
  return head == NULL;
}
```

```c
#include "STACK.h"

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

Allocate memory and initialize new node

19

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) {
    head = NEWnode(item, head);
}

20

int STACKpop(void) {
    int item;
    if (head == NULL) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
    item = head->item;
    link x = head->next;
    free(head);
    head = x;
    return item;
}

void STACKshow(void) {
    link x;
    for (x = head; x != NULL; x = x->next)
        printf("%d\n", x->item);
}

Stack Implementation With Linked Lists

implement memory and free new node
insert at beginning of list

21

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

22

Conclusions

Whew, lots of material in this lecture!

Pointers are useful, but confusing.

Study these slides and carefully read relevant material.
# Lecture P8: Extra Slides

## 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;
}
```

### “Pointer arithmetic”

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

just to stress that `a[i]` really means `*(a+i)`:

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

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

## 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;
}
```

```c
int main(void) {
    int a[N] = {84, 67, 24, ..., 89, 90};
    printf("%d\n", average(a, N));
    return 0;
}
```
Why Pass Array as Pointer?

Advantages.

1. Efficiency for large arrays – don’t want to copy entire array.
2. 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;
}

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

Passing Arrays to Functions

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

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

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