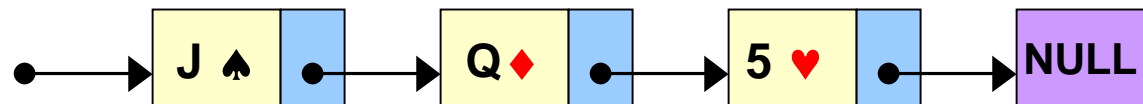


Lecture P8: Pointers and Linked Lists



Pointer Overview

Basic computer memory abstraction.

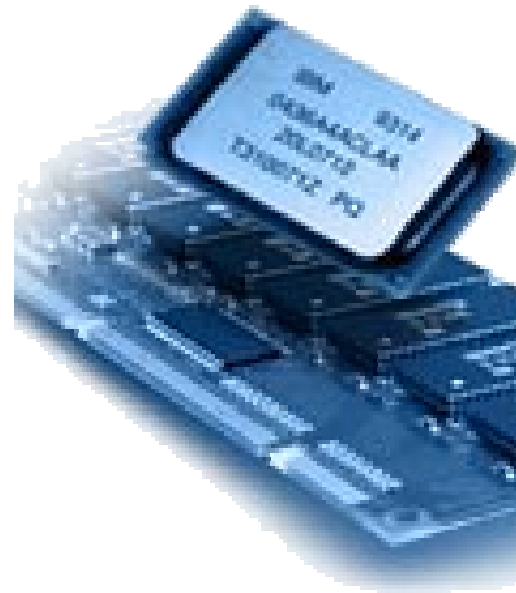
- Indexed sequence of bits.
- Address = index.

Pointer = **VARIABLE** that stores memory address.

Uses.

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

addr	value
0	0
1	1
2	1
3	1
4	0
5	1
6	0
7	0
8	1
9	0
10	1
...	...
256GB	1

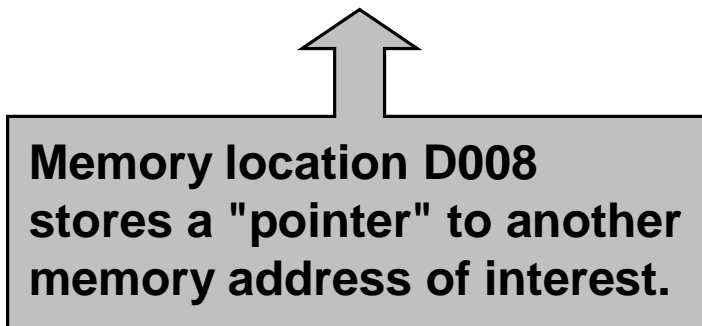


Pointers in TOY

Variable that stores the value of a single MEMORY ADDRESS.

- In TOY, memory addresses are 00 – FF.
 - **indexed addressing: store a memory address in a register**
- Very powerful and useful programming mechanism.
- Confusing and easy to abuse!

Address	D000	D004	D008	..	D0C8	D0CC	D0D0	..	D200	D204	D208
Value	9	1	D200	..	0	7	0000	..	5	3	D0C8



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

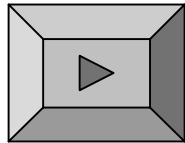
Pointers in C

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
```

allocate storage for
pointer to int



```
pointer.c
#include <stdio.h>

int main(void) {
  int x;
  int *px;

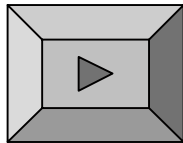
  x = 7;
  px = &x;
  printf("  x = %d\n");
  printf(" px = %p\n", px);
  printf("*px = %d\n", *px);
  return 0;
}
```

Pointers as Arguments to Functions

Goal: function that swaps values of two integers.

A first attempt:

only swaps copies
of x and y



badswap.c

```
#include <stdio.h>

void swap(int a, int b) {
    int t;
    t = a; a = b; b = t;
}

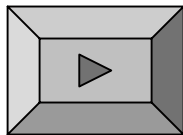
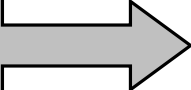
int main(void) {
    int x = 7, y = 10;
    swap(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

Pointers as Arguments to Functions

Goal: function that swaps values of two integers.

Now, one that works.

changes value
stored in memory
address for x and y



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;
}
```

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.
- ...

addr	value
0	0
1	1
2	1
3	1
4	0
5	1
6	0
7	0
8	1
9	0
10	1
...	...
256GB	1



Linked List

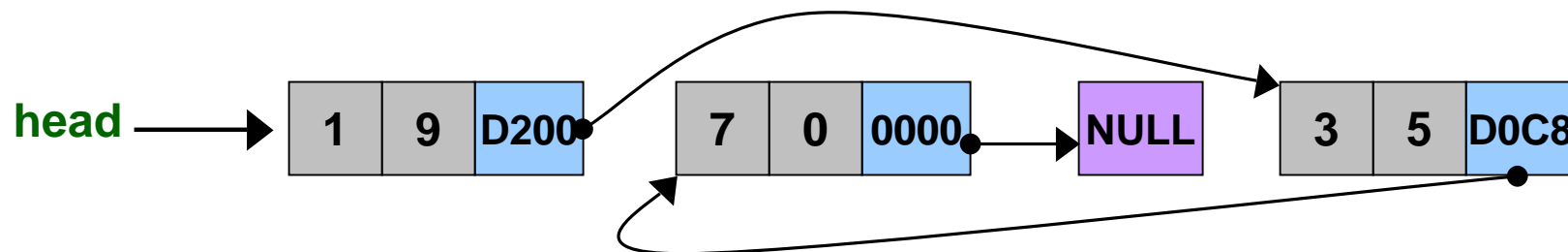
Fundamental data structure.

- 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 memory representation of $x^9 + 3x^5 + 7$.

- Assume linked list starts in location D000.

Address	D000	D004	D008	..	D0C8	D0CC	D0D0	..	D200	D204	D208
Value	1	9	D200	..	7	0	0000	..	3	5	D0C8



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**.

Address	D000	D004	D008	..	D0C8	D0CC	D0D0	..	D200	D204	D208
Value	1	9	D200	..	7	0	0000	..	3	5	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. $x^7 + x^6 + 3x^4 + 2x^3 + 1$

Huge Sparse Polynomial		
	array	linked
space	huge	tiny
time	instant	tiny

Huge Dense Polynomial		
	array	linked
space	huge	3 * huge
time	instant	huge

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^k .

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.

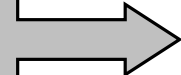
- `->` and `.` operators

Linked List for Polynomial

C code to represent
of $x^9 + 3x^5 + 7$.

- Statically, using nodes.

memory address
of next node



initialize data



link up nodes



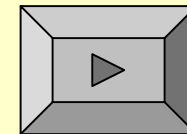
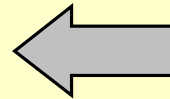
poly1.c

```
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;
}
```

define node to
store two integers



Linked List for Polynomial

C code to represent
of $x^9 + 3x^5 + 7$.

- Statically, using nodes.
- Dynamically using links.

initialize data

allocate enough
memory to store node

link up nodes of list

poly2.c

```
#include <stdlib.h>

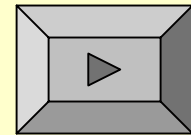
typedef struct node *link;
struct node { . . . };

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;
}
```



Study this code: tip of iceberg!

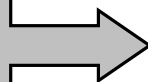
Better Programming Style

Write separate function to handle memory allocation and initialization.

check if malloc fails



Initialize pointers to NULL



```
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 = 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.

STACK.h

```
void STACKinit(void);  
int  STACKisempty(void);  
void STACKpush(int);  
int  STACKpop(void);  
void STACKshow(void);
```

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

client.c

```
#include "STACK.h"  
  
int main(void) {  
    int a, b;  
    . . .  
    STACKinit();  
    STACKpush(a);  
    . . .  
    b = STACKpop();  
    return 0;  
}
```

Stack Implementation With Linked Lists

stacklist.c

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

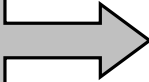
typedef struct STACKnode* link;
struct STACKnode {
    int item;
    link next;
};

static link head = NULL;

void STACKinit(void) {
    head = NULL;
}

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

static to make
it a true ADT



standard linked
list data structure



head points to
top node on stack



Stack Implementation With Linked Lists

allocate memory and
initialize new node



stacklist.c (cont)

```
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);  
}
```

insert at beginning
of list

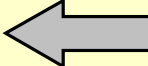


Stack Implementation With Linked Lists

stacklist.c (cont)

```
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;
}
```

free is opposite of malloc:
gives memory back to system



traverse linked list



```
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
```

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.

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;
}
```

on arizona,
int is 32 bits (4 bytes) ⇒
4 byte offset

"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

Memory address	D000	D004	D008	..	D0F8	D0FC	..
Value	84	67	24	..	89	90	..

Pointers and Arrays

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!!!

integer (on arizona) takes 4 bytes \Rightarrow 4 byte offset

"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`

Memory address	D000	D004	D008	..	D0F8	D0FC	..
Value	84	67	24	..	89	90	..

Passing Arrays to Functions

Pass array to function.

- Pointer to array element 0 is passed instead.

```
avg.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;
}
```

receive the value D000 from main

passes &a[0] = D000 to function

Why Pass Array as Pointer?

Advantages.



```
avg.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);
    . . .
}
```

compute average of
a[5] through a[14]

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.

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;  
}
```