6.087 Lecture 7 – January 20, 2010

- Review
- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays
- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

Review: Compound data types

- **struct** - structure containing one or multiple fields, each with its own type (or compound type)
  - size is combined size of all the fields, padded for byte alignment
  - anonymous or named
- **union** - structure containing one of several fields, each with its own type (or compound type)
  - size is size of largest field
  - anonymous or named
- Bit fields - structure fields with width in bits
  - aligned and ordered in architecture-dependent manner
  - can result in inefficient code

Review: Compound data types

- Consider this compound data structure:

```c
struct foo {
  short s;
  union {
    int i;
    char c;
  } u;
  unsigned int flag_s : 1;
  unsigned int flag_u : 2;
  unsigned int bar;
};
```

- Assuming a 32-bit x86 processor, evaluate
  `sizeof(struct foo)`

Review: Compound data types

- Consider this compound data structure:

```
struct foo {
  short s;                         ← 2 bytes
  union {                          ← 4 bytes,
    int i;                         4 byte-aligned
    char c;
  } u;
  unsigned int flag_s : 1;         ← bit fields
  unsigned int flag_u : 2;
  unsigned int bar;                ← 4 bytes,
};                                 4 byte-aligned
```

- Assuming a 32-bit x86 processor, evaluate
  sizeof(struct foo)

Review: Compound data types

- How can we rearrange the fields to minimize the size of `struct foo`?

Review: Compound data types

- How can we rearrange the fields to minimize the size of **struct** foo?
- Answer: order from largest to smallest:

```c
struct foo {
  union {
    int i;
    char c;
  } u;
  unsigned int bar;
  short s;
  unsigned int flag_s : 1;
  unsigned int flag_u : 2;
};
```

sizeof(**struct** foo) $= 12$

Review: Linked lists and trees

- Linked list and tree dynamically grow as data is added/removed
- Node in list or tree usually implemented as a **struct**
- Use `malloc()`, `free()`, etc. to allocate/free memory dynamically
- Unlike arrays, do not provide fast random access by index (need to iterate)

- Review
- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays
- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

Pointer review

- Pointer represents address to variable in memory
- Examples:

  int $*$pn; – pointer to `int`

  struct div_t $*$ pdiv; – pointer to structure `div_t`
- Addressing and indirection:

```
double pi = 3.14159;
double *ppi = &pi;
printf("pi = %g\n", *ppi);
```

- Today: pointers to pointers, arrays of pointers, multidimensional arrays

Pointers to pointers

- Address stored by pointer also data in memory
- Can address location of address in memory – pointer to that pointer

```
int n = 3;
int *pn = &n; /* pointer to n */
int **ppn = &pn; /* pointer to address of n */
```

- Many uses in C: pointer arrays, string arrays

Pointer pointers example

- What does this function do?

```
void swap(int **a, int **b) {
  int *temp = *a;
  *a = *b;
  *b = temp;
}
```

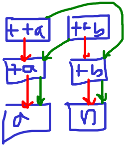
sawping the first pointer address
which means that

or
----
int *temp;
temp = *a;

Pointer pointers example

- What does this function do?

```c
void swap(int **a, int **b) {
  int *temp = *a;
  *a = *b;
  *b = temp;
}
```

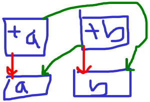


before

after

- How does it compare to the familiar version of swap?

```c
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```



before

after

Pointer arrays

- Pointer array – array of pointers
  `int *arr[20];` – an array of pointers to `int`'s
  `char *arr[10];` – an array of pointers to `char`'s
- Pointers in array can point to arrays themselves
  `char *strs[10];` – an array of `char` arrays (or strings)

Pointer array example

- Have an array `int arr[100];` that contains some numbers
- Want to have a sorted version of the array, but not modify `arr`
- Can declare a pointer array `int * sorted_array[100];` containing pointers to elements of `arr` and sort the pointers instead of the numbers themselves
- Good approach for sorting arrays whose elements are very large (like strings)

Pointer array example

Insertion sort:

```
/* move previous elements down until
   insertion point reached */
void shift_element(unsigned int i) {
  int *pvalue;
  /* guard against going outside array */
  for (pvalue = sorted_array[i]; i &&
         *sorted_array[i-1] > *pvalue; i--) {
    /* move pointer down */
    sorted_array[i] = sorted_array[i-1];
  }
  sorted_array[i] = pvalue; /* insert pointer */
}
```

?

Pointer array example

Insertion sort (continued):

```
/* iterate until out-of-order element found;
   shift the element, and continue iterating */
void insertion_sort(void) {
  unsigned int i, len = array_length(arr);
  for (i = 1; i < len; i++)
    if (*sorted_array[i] < *sorted_array[i-1])
      shift_element(i);
}
```

?

String arrays

- An array of strings, each stored as a pointer to an array of chars
- Each string may be of different length

```
char str1[] = "hello"; /* length = 6 */
char str2[] = "goodbye"; /* length = 8 */
char str3[] = "ciao"; /* length = 5 */
char * strArray[] = {str1, str2, str3};
```

- Note that strArray contains only pointers, not the characters themselves!

Multidimensional arrays

- C also permits multidimensional arrays specified using `[ ]` brackets notation:

  `int world[20][30];` is a 20x30 2-D array of `int`'s
- Higher dimensions possible:

  `char bigcharmatrix [15][7][35][4];` – what are the dimensions of this?
- Multidimensional arrays are rectangular; pointer arrays can be arbitrary shaped

6.087 Lecture 7 – January 20, 2010

- Review
- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays
- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

More data structures

- Last time: linked lists
- Today: stack, queue
- Can be implemented using linked list or array storage

The stack

- Special type of list - last element in (push) is first out (pop)
- Read and write from same end of list
- The stack (where local variables are stored) is implemented as a *gasp* stack

Stack as array

- Store as array buffer (static allocation or dynamic allocation):

```
int stack_buffer[100];
```

- Elements added and removed from end of array; need to track end:

```
int itop = 0; /* end at zero => initialized for empty stack */
```

Stack as array

- Add element using **void** push(**int**);

```
void push(int elem) {
  stack_buffer[itop++] = elem;
}
```

push

- Remove element using **int** pop(**void**);

```
int pop(void) {
  if (itop > 0)
    return stack_buffer[--itop];
  else
    return 0; /* or other special value */
}
```

POP

- Some implementations provide **int** top(**void**); to read last (top) element without removing it

Stack as linked list

- Store as linked list (dynamic allocation):

```
struct s_listnode {
  int element;
  struct s_listnode * pnext;
};
```

`struct s_listnode * stack_buffer = NULL;` – start empty

- "Top" is now at front of linked list (no need to track)

Stack as linked list

- Add element using **void** push(**int**);

```c
void push(int elem) {
  struct s_listnode *new_node = /* allocate new node */
    (struct s_listnode *)malloc(sizeof(struct s_listnode))
  new_node->pnext = stack_buffer;
  new_node->element = elem;
  stack_buffer = new_node;
}
```

- Adding an element pushes back the rest of the stack

Stack as linked list

- Remove element using `int pop(void);`

```
int pop(void) {
  if (stack_buffer) {
    struct s_listnode *pelem = stack_buffer;
    int elem = stack_buffer->element;
    stack_buffer = pelem->pnext;
    free(pelem); /* remove node from memory */
    return elem;
  } else
    return 0; /* or other special value */
}
```

- Some implementations provide `int top(void);` to read last (top) element without removing it

The queue

- Opposite of stack - first in (enqueue), first out (dequeue)
- Read and write from opposite ends of list
- Important for UIs (event/message queues), networking (Tx, Rx packet queues)
- Imposes an ordering on elements

Queue as array

- Again, store as array buffer (static or dynamic allocation);

```
float queue_buffer[100];
```

- Elements added to end (rear), removed from beginning (front)
- Need to keep track of front and rear:

```
int ifront = 0, irear = 0;
```

- Alternatively, we can track the front and number of elements:

```
int ifront = 0, icount = 0;
```

- We'll use the second way (reason apparent later)

Queue as array

- Add element using **void** enqueue(**float**);

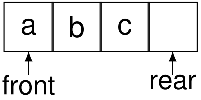
```
void enqueue(float elem) {
  if (icount < 100) {
    queue_buffer[ifront+icount] = elem;
    icount++;
  }
}
```

- Remove element using **float** dequeue(**void**);

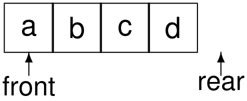
```
float dequeue(void) {
  if (icount > 0) {
    icount--;
    return queue_buffer[ifront++];
  } else
    return 0.; /* or other special value */
}
```

Queue as array

- This would make for a very poor queue! Observe a queue of capacity 4:



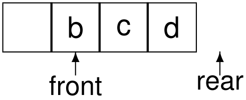
- Enqueue `'d'` to the rear of the queue:



The queue is now full.

Queue as array

- Dequeue `'a'`:



- Enqueue `'e'` to the rear: where should it go?
- Solution: use a circular (or "ring") buffer
  - `'e'` would go in the beginning of the array

Queue as array

- Need to modify **void** enqueue(**float**); and **float** dequeue(**void**);
- New **void** enqueue(**float**);:

```
void enqueue(float elem) {
  if (icount < 100) {
    queue_buffer[(ifront+icount) % 100] = elem;
    icount++;
  }
}
```

Queue as array

- New **float** dequeue(**void**);:

```
float dequeue(void) {
  if (icount > 0) {
    float elem = queue_buffer[ifront];
    icount--;
    ifront++;
    if (ifront == 100)
      ifront = 0;
    return elem;
  } else
    return 0.; /* or other special value */
}
```

- Why would using “front” and “rear” counters instead make this harder?

Queue as linked list

- Store as linked list (dynamic allocation):

```
struct s_listnode {
  float element;
  struct s_listnode * pnext;
};
```

```
struct s_listnode *queue_buffer = NULL;
```
– start empty

- Let front be at beginning – no need to track front
- Rear is at end – we should track it:

```
struct s_listnode *prear = NULL;
```

Queue as linked list

- Add element using `void enqueue(float);`

```
void enqueue(float elem) {
  struct s_listnode *new_node = /* allocate new node */
    (struct s_listnode *)malloc(sizeof(struct s_listnode));
  new_node->element = elem;
  new_node->pnext = NULL; /* at rear */
  if (prear)
    prear->pnext = new_node;
  else /* empty */
    queue_buffer = new_node;
  prear = new_node;
}
```

- Adding an element doesn't affect the front if the queue is not empty

Queue as linked list

- Remove element using **float** dequeue(**void**);

```
float dequeue(void) {
  if (queue_buffer) {
    struct s_listnode *pelem = queue_buffer;
    float elem = queue_buffer->element;
    queue_buffer = pelem->pnext;
    if (pelem == prear) /* at end */
      prear = NULL;
    free(pelem); /* remove node from memory */
    return elem;
  } else
    return 0.; /* or other special value */
}
```

- Removing element doesn't affect rear unless resulting queue is empty

A simple calculator

- Stacks and queues allow us to design a simple expression evaluator
- Prefix, infix, postfix notation: operator before, between, and after operands, respectively

Infix	Prefix	Postfix
A + B	+ A B	A B +
A * B - C	- * A B C	A B * C -
(A + B) * (C - D)	* + A B - C D	A B + C D - *

- Infix more natural to write, postfix easier to evaluate

Infix to postfix

- "Shunting yard algorithm" - Dijkstra (1961): input and output in queues, separate stack for holding operators
- Simplest version (operands and binary operators only):
  1. dequeue token from input
  2. if operand (number), add to output queue
  3. if operator, then pop operators off stack and add to output queue as long as
     - top operator on stack has higher precedence, or
     - top operator on stack has same precedence and is left-associative

     and push new operator onto stack
  4. return to step 1 as long as tokens remain in input
  5. pop remaining operators from stack and add to output queue

Infix to postfix example

- Infix expression: A + B * C - D

Token	Output queue	Operator stack
A	A	
+	A	+
B	A B	+
*	A B	+ *
C	A B C	+ *
-	A B C * +	-
D	A B C * + D	-
(end)	A B C * + D -	

- Postfix expression: A B C * + D -
- What if expression includes parentheses?

Example with parentheses

- Infix expression: (A + B) * (C - D)

Token	Output queue	Operator stack
(		(
A	A	(
+	A	(+
B	A B	(+
)	A B +	
*	A B +	*
(	A B +	* (
C	A B + C	* (
-	A B + C	* (-
D	A B + C D	* (-
)	A B + C D -	*
(end)	A B + C D - *	

- Postfix expression: A B + C D - *

Evaluating postfix

- Postfix evaluation very easy with a stack:
  1. dequeue a token from the postfix queue
  2. if token is an operand, push onto stack
  3. if token is an operator, pop operands off stack (2 for binary operator); push result onto stack
  4. repeat until queue is empty
  5. item remaining in stack is final result

Postfix evaluation example

- Postfix expression: 3 4 + 5 1 - *

Token	Stack
3	3
4	3 4
+	7
5	7 5
1	7 5 1
-	7 4
*	28
(end)	answer $= 28$

- Extends to expressions with functions, unary operators
- Performs evaluation in one pass, unlike with prefix notation

Summary

Topics covered:

- Pointers to pointers
  - pointer and string arrays
  - multidimensional arrays
- Data structures
  - stack and queue
  - implemented as arrays and linked lists
  - writing a calculator

6.087 Practical Programming in C
January (IAP) 2010