Pointers to Functions and Table Lookup
Complicated Declarations

- `char **argv;`
  - argv: pointer to pointer to char

- `int (*daytab)[13];`
  - daytab: pointer to array[13] of int

- `int *daytab[13];`
  - daytab: array[13] of pointer to int
Complicated Declarations

- `void *comp();`
  - `comp`: function returning pointer to `void`

- `void (*comp)();`
  - `comp`: pointer to function returning `void`

- `void (*comp[3])(());`
  - `comp`: array[3] of pointer to function returning `void`
Complicated Declarations

- `void ((*comp[3])(()))[5];`
  - Error! Why?

- `void (*(*comp[3])(()))[5];`
  - Error! (declaration of 'comp' as `array of voids`)
Complicated Declarations

- \texttt{char (*(*x()))[]());}
  - \texttt{x: function returning pointer to array[]} of \texttt{pointer to function returning char}

- \texttt{char (*(*x[3]))()[5];}
  - \texttt{x: array[3]} of \texttt{pointer to function returning pointer to array[5]} of \texttt{char}
char *lineptr[MAXLINES];

qsort((void **)lineptr, 0, nlines-1, (int (*)(void *, void *))((numeric ? numcmp : strcmp)));

int numcmp(char *s1, char *s2)
{
    double v1, v2;
    v1 = atof(s1);  v2 = atof(s2);
    if (v1<v2) return -1;
    else if (v1>v2) return 1;
    else return 0;
}
void qsort(void *v[], int left, int right, int (*comp)(void *, void *))
{
    int i, last;
    if (left >= right) return;
    swap(v, left, (left+right)/2);
    last=left;
    for (i=left+1; i<=right; i++)
        if (comp(v[i], v[left])<0) swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
Table Lookup

struct nlist { /*table entry: */
    struct nlist *next; /* next entry in chain */
    char *name; /* defined name */
    char *defn; /* replacement text */
};

#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* pointer table */
Table Lookup

/* hash: form hash value for string s */
unsigned hash(char *s)
{
    unsigned hashval;
    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
/* lookup: look for s in hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;
    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np; /* found */

    return NULL; /* not found */
}
Table Lookup

```c
struct nlist *lookup(char *);
char *strdup(char *);

/* install: put (name, defn) in hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;
    if ((np = lookup(name)) == NULL) { /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else /* already there */
        free((void *) np->defn); /*free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}

char *strdup(char *s) /* make a duplicate of s */
{
    char *p;
    p = (char *) malloc(strlen(s)+1); /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```
Binary Search Tree

```
  now
  /   /
is   the
  /
for men of time
  /
all good party their to
  /
aid come
```
struct tnode {
    /* the tree node: */
    char *word;    /* points to the text */
    int count;     /* number of occurrences */
    struct tnode *left; /* left child */
    struct tnode *right; /* right child */
};
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);
/* word frequency count */
main()
{
    struct tnode *root;
    char word[MAXWORD];
    root = NULL;
    while (getword(word, MAXWORD) != EOF)
    {
        if (isalpha(word[0]))
            root = addtree(root, word);
    }
    treeprint(root);
    return 0;
}
```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree: add a node with w, at or below p */
struct treenode *addtree(struct tnode *p, char *w)
{
    int cond;
    if (p == NULL) { /* a new word has arrived */
        p = talloc(); /* make a new node */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++; /* repeated word */
    else if (cond < 0) /* less than into left subtree */
        p->left = addtree(p->left, w);
    else /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}
Questions?