Today's Lecture

1. Questions on Programming Assignment I?

2. **Code Walk of `sh.c`**

   (a) High-level Control Structure

   (b) `#defines`

   (c) Important Declarations

   (d) Not-so-important Declarations

   (e) Functions
   - `int main(int argc, char *argv[], char *envp[])`
   - parsing input
   - redirecting IO
   - executing commands
   - miscellaneous

3. Extending set of built-in commands in `sh.c`

sh.c: High-level Control Structure

Basic shell loop:

```
forever {
    print prompt
    read a command
    execute it
}
```

A bit more realistic:

```
forever {
    print prompt

    parse next input command into a parse tree
       (each node has info about one operation)
       (simple command, pipe, redirection, etc.)

    walk the tree to execute the components
       at each node
            create subprocesses
            do redirection and plumbing
            execute the commands
}
```


sh.c: Sample Session


```
: arran 180; ls
sh.c
: arran 181; gcc -ansi sh.c
: arran 182; ls
a.out*  sh.c
: arran 183; a.out
$ ls
a.out   sh.c
$ ls >dir-files
$ ls
a.out           dir-files       sh.c
$ cat dir-files
a.out
dir-files
sh.c
$ wc dir-files
       3       3      21 dir-files
$ exec a.out
$ (ls | wc) >>dir-files
$ cat <dir-files
a.out
dir-files
sh.c
       3       3      21
$ asdf
asdf: not found (2)
$ rm dir-files
$ ls >dir-files &
14187
$ a.out
$
$ exit
$ exit
: arran 184;
```


sh.c: Parsing Functions

Functions for parsing input:

- `struct node *alloc(int (*op)())`

- `struct node *command()`
- `struct node *simple()`
- `struct node *list()`
- `struct node *pipeline()`

- `int getc()`
- `int gettoken()`

`sh.c`: Execution Functions

Functions for executing parsed commands:

- `int xwait(struct node *np)`
- `int xnowait(struct node *np)`
- `int builtin(struct node *np)`
- `int xpipeline(struct node *np)`
- `int xsimple(struct node *np)`
- `int xsubshell(struct node *np)`

`sh.c`: IO Redirection Functions

Functions used in redirecting IO:

- `int setio(struct node *np)`
- `int redirect(struct node *np)`
- also `open`, `close`, `creat`, `lseek`, `pipe`, `dup`

`sh.c`: Miscellaneous Functions

- `int strcmp(char *s1, char *s2)`
- `int perror(char *s, char *t)`

`sh.c`: CPP directives


```
#define MAXLINE 200      /* maximum line length */
#define WORD 256         /* token code for words */
#define EOF -1           /* token code for end of file */
#define ispunct(c)       (c=='|' || c=='&' || c==';' || c=='<' || \
                          c=='>' || c=='(' || c==')' || c=='\n')
#define isspace(c)       (c==' ' || c=='\t')
#define execute(np)      (np ? (*(np)->op)(np) : 0)
```

sh.c: Important Declarations


```
struct node {                   /* parse tree node */
        int (*op)();            /* operator function */
        struct node *args[2];   /* argument nodes */
        char *argv[100];        /* argument pointers */
        char *io[3];            /* i/o redirection */
} nodes[100];                   /* node pool */

struct node *nfree;             /* next available node */


char strspace[10*MAXLINE];      /* string storage */
char *sfree;                    /* next free char in strspace */

int t;                          /* current token code */
char *token;                    /* current token text (in strspace) */

int putback = 0;                /* lookahead */

int status = 0;                 /* exit status of most recent cmd */

char **env;                     /* environment of caller */

char *path[] = {".", "/bin", "/usr/bin", "/usr/ucb", 0};
```


sh.c: Not-so-important Declarations


```
extern int errno;

struct node *alloc(), *list(), *command(), *simple(), *pipeline();

int xsimple(), xpipeline(), xwait(), xnowait(), xsubshell();
```


sh.c: main


```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
{
        struct node *np;

        env = envp;
        for (;;) {
                write(2, "$ ", 2);
                nfree = nodes;
                sfree = strspace;
                if ((t = gettoken()) == EOF) break;
                if (t != '\n')
                        if (np = list()) execute(np);
                        else perror("syntax error", "");
                while (t != EOF && t != '\n')   /* flush syntax errors */
                        t = gettoken();
                }
        exit(status);
}
```


sh.c: alloc


```
/* alloc - allocate for op and return a node */

struct node *alloc(op)
int (*op)();
{
        if (nfree < nodes + sizeof(nodes)) {
                nfree->op = op;
                nfree->args[0] = nfree->args[1] = 0;
                nfree->argv[0] = nfree->argv[1] = 0;
                nfree->io[0] = nfree->io[1] = nfree->io[2] = 0;
                return nfree++;
                }
        perror("node storage overflow", "");
        exit(2);
}
```

```
/* command - ( list ) [ ( < | > | >> ) word ]* | simple */

struct node *command()
{
        struct node *np;

        if (t != '(')
                return simple();
        np = alloc(xsubshell);
        t = gettoken();
        if (!(np->args[0] = list()) || t != ')') return 0;
        while ((t = gettoken()) == '<' || t == '>')
                if (!setio(np)) return 0;
        return np;
}
```



```
/* list - pipeline ( ( ; | & ) pipeline )* [ ; | & ]  (not LL(1), but ok) */

struct node *list()
{
        struct node *np, *np1;

        np = alloc((int (*)()) 0);
        if (!(np->args[1] = pipeline())) return 0;
        while (t == ';' || t == '&') {
                np->op = (t == ';') ? xwait : xnowait;
                t = gettoken();
                if (t == ')' || t == '\n')      /* tests ~first(pipeline) */
                        break;
                np1 = alloc((int (*)()) 0);
                np1->args[0] = np;
                if (!(np1->args[1] = pipeline())) return 0;
                np = np1;
                }
        if (np->op == 0) np->op = xwait;
        return np;
}
```



```
/* pipeline - command ( | command )* */

struct node *pipeline()
{
        struct node *np, *np1;

        if (!(np = command())) return 0;
        while (t == '|') {
                np1 = alloc(xpipeline);
                np1->args[0] = np;
                t = gettoken();
                if (!(np1->args[1] = command())) return 0;
                np = np1;
                }
        return np;
}
```



```
/* simple - word ( [ < | > | >> ] word )* */

struct node *simple()
{
        struct node *np;
        int n = 1;

        if (t != WORD) return 0;
        np = alloc(xsimple);
        np->argv[0] = token;
        while ((t = gettoken()) == WORD || t == '<' || t == '>')
                if (t == WORD) np->argv[n++] = token;
                else if (!setio(np)) return 0;
        np->argv[n] = 0;
        return np;
}
```

sh.c: setio


```
/* setio - ( < | > | >> ) word; fill in np->io[] */

int setio(np)
struct node *np;
{
	if (t == '<') { t = gettoken(); np->io[0] = token; }
	else if (t == '>') {
		t = gettoken();
		if (t == '>') { t = gettoken(); np->io[2] = token; }
		else np->io[1] = token;
		}
	else return 0;
	if (t != WORD) return 0;
	return 1;
}
```


sh.c: redirect


```
/* redirect - redirect i/o according to np->io[] values */

redirect(np)
struct node *np;
{
	if (np->io[0]) {
		close(0);
		if (open(np->io[0], 0) != 0)
			{ perror(": can't open", np->io[0]); exit(1); }
		}
	if (np->io[1]) {
		close(1);
		if (creat(np->io[1], 0644) != 1)
			{ perror(": can't create", np->io[1]); exit(1); }
		}
	if (np->io[2]) {
		close(1);
		if (open(np->io[2], 1) != 1 && creat(np->io[2], 0644) != 1)
			{ perror(": can't write", np->io[2]); exit(1); }
		lseek(1, 0L, 2);
		}
}
```


sh.c: builtin


```
/* builtin - check np for builtin command and, if found, execute it */

int builtin(np)
struct node *np;
{
	int pid, n = 0;
	char *s, name[MAXLINE];

	if (s = np->argv[1])
		while (*s >= '0' && *s <= '9')
			n = 10*n + *s++ - '0';
	if (strcmp(np->argv[0], "cd") == 0) {
		if (np->argv[1] && chdir(np->argv[1]) == -1)
			perror(": bad directory", np->argv[0]);
		return 1;
		}
	else if (strcmp(np->argv[0], "exit") == 0)
		exit(np->argv[1] ? n : status);
	else if (strcmp(np->argv[0], "wait") == 0) {
		if (n > 0)
			while ((pid = wait(&status)) > 0 && pid != n)
			    ;
		else
			wait(0);
		return 1;
		}
	else if (strcmp(np->argv[0], "exec") == 0) {
		redirect(np);
		if (np->argv[1] == (char *) 0)
			return 1;
		for (n = 0; path[n]; n++) {
			sprintf(name, "%s/%s", path[n], np->argv[1]);
			execve(name, &np->argv[1], env);
			}
		perror(": not found", np->argv[1]);
		return 1;
		}
	return 0;
}
```


sh.c: xpipeline


```
/* xpipeline - execute cmd | cmd */

int xpipeline(np)
struct node *np;
{
	int pid, n, fd[2];

	if (pipe(fd) < 0)
		{ perror("can't create pipe", ""); return 0; }
	if ((pid = fork()) == 0) {	/* left side; redirect stdout */
		close(1);
		dup(fd[1]);
		close(fd[0]);
		close(fd[1]);
		execute(np->args[0]);
		exit(status);
		}
	else if (pid == -1)
		{ perror("can't create process", ""); return 0; }
	if ((pid = fork()) == 0) {	/* right side; redirect stdin */
		close(0);
		dup(fd[0]);
		close(fd[0]);
		close(fd[1]);
		if ((pid = execute(np->args[1])) > 0) {
			while ((n = wait(&status)) > 0 && pid != n) ;
			if (n == -1) perror("wait error", "");
			}
		exit(status);
		}
	else if (pid == -1)
		{ perror("can't create process", ""); return 0; }
	close(fd[0]);	/* avoid using up fd's */
	close(fd[1]);
	return pid;
}
```

```
/* xsimple - execute a simple command */

int xsimple(np)
struct node *np;
{
        char name[MAXLINE];
        int pid, i;

#ifdef COS333_P1.2
        if (var_assignment(np)) return 0;
#endif
        if (builtin(np)) return 0;
        if (pid = fork()) {
                if (pid == -1)
                        perror(": can't create process", np->argv[0]);
                return pid;
                }
        redirect(np);   /* child process */
        if (*np->argv[0] == '/' || *np->argv[0] == '.')
                execve(np->argv[0], np->argv, env);
        else
                for (i = 0; path[i]; i++) {
                        sprintf(name, "%s/%s", path[i], np->argv[0]);
                        execve(name, np->argv, env);
                        }
        perror(": not found", np->argv[0]);
        exit(2);
}
```



```
/* xsubshell - execute (cmd) */

int xsubshell(np)
struct node *np;
{
        int pid;

        if (pid = fork()) {
                if (pid == -1) perror("can't create process", "");
                return pid;
                }
        redirect(np);   /* child process */
        execute(np->args[0]);
        exit(status);
}
```



```
/* xnowait - execute cmd & */

int xnowait(np)
struct node *np;
{
        char *s, buf[MAXLINE];

        execute(np->args[0]);
        sprintf(s = buf, "%5d\n", execute(np->args[1]));
        while (*s == ' ') s++;
        write(2, s, buf + 6 - s);
        return 0;
}
```



```
/* xwait - execute cmd ; */

int xwait(np)
struct node *np;
{
        int n, pid;

        execute(np->args[0]);
        if ((pid = execute(np->args[1])) > 0) {
                while ((n = wait(&status)) > 0 && n != pid) ;
                if (n == -1) perror("wait error", "");
                }
        return 0;
}
```

```
/* getc - get next, possibly pushed back, input character */

int getc()
{
        char c;

        if (putback) { c = putback; putback = 0; }
        else if (read(0, &c, 1) != 1) return EOF;
        return c;
}
```



```
/* gettoken - get next token into string space, return token code */
int gettoken()
{
        int c;

        while ((c = getc()) != EOF) if (!isspace(c)) break;
        if (c == EOF || ispunct(c)) return c;
        token = sfree;
        do {
                if (sfree >= strspace + sizeof(strspace) - 1)
                        { perror("string storage overflow", ""); exit(2); }
                *sfree++ = c;
                } while ((c = getc()) != EOF && !ispunct(c) && !isspace(c));
        *sfree++ = 0;
        putback = c;
#ifdef COS333_P1.2
        expand(token);
#endif
        return WORD;
}
```



```
/* perror - print error message s, prefixed by t */

perror(s, t)
char *s, *t;
{
        char buf[MAXLINE];

        if (errno > 0) sprintf(buf, "%s%s (%d)\n", t, s, errno);
        else sprintf(buf, "%s%s\n", t, s);
        for (s = buf; *s; s++) ;
        write(2, buf, s - buf);
        errno = 0;
}
```



```
/* strcmp - compare s1 and s2 */

int strcmp(s1, s2)
char *s1, *s2;
{
        while (*s1 == *s2++) if (*s1++ == 0) return 0;
        return *s1 - *--s2;
}
```

sh.c: One Implementation of print

To correctly redirect IO, a `print` command can create a child process:

```
/* ... in function builtin */

else if (strcmp(np->argv[0], "print") == 0) {
  int pid;

  if ((pid = fork()) == -1)
    perror(": can't create process", np->argv[0]);
  if (pid == 0) {
    int i;

    redirect(np);        /* child process */
    for (i = 1; np->argv[i]; i++)
      printf("%s ", np->argv[i]);
    putchar('\n');
    exit(0);
  } else {
    int n, status;

    while ((n = wait(&status)) > 0 && n != pid) ;
    if (n == -1) perror(": wait error", "print: ");
  }
  return 1;
}

/* rest of function builtin ... */
```


sh.c: Another Implementation of print

Or, a `print` command can explicitly redirect process:

```
/* ... function builtin */

else if (strcmp(np->argv[0], "print") == 0) {
  int fd;
  int i;

  if (np->io[1]) {
    if ((fd = open(np->io[1],
                   O_WRONLY|O_CREAT|O_TRUNC,
                   S_IRUSR|S_IWUSR)) == -1)
      perror(": can't redirect output", np->argv[0]);
  } else if (np->io[2]) {
    if ((fd = open(np->io[1],
                   O_WRONLY|O_APPEND|O_CREAT,
                   S_IRUSR|S_IWUSR)) == -1)
      perror(": can't redirect  appended ouput", np->argv[0]);
  } else fd = 1;
  for (i = 1; np->argv[i]; i++) {
    write(fd, np->argv[i], strlen(np->argv[i]));
    write(fd, " ", 1);
  }
  write(fd, "\n", 1);
  if (fd != 1) close(fd);
  return 1;
}

/* rest of function builtin ... */
```

Note: must `#include <string.h>` and `<sys/fcntl.h>`

Next Lecture

- Run `~cs333/bin/ksh`
- Read `man` pages for:

  `grep`, `egrep`

  `ed` (section on regular expressions)

  `sed`
- We'll talk about a sample project