```
:::::::::::::: sneakysh.c ::::::::::::::

#include	<stdio.h>
/*
 *	a sneaky way of writing a shell
 */

main()
{
	char	command_line[BUFSIZ];

	while( printf("$ ") != EOF && gets(command_line) )
		system( command_line );
}

:::::::::::::: execdemo1.c ::::::::::::::

main()
{
	char *args[4];

	args[0] = "ls";				/* build the arglist */
	args[1] = "-l";
	args[2] = "/usr/bin";
	args[3] = (char *) 0;		/* terminate with NULL */

	printf("about to execute ls -l /usr/bin\n" );
	execvp( "ls" , args );
	printf( "that was ls, cool, eh?\n");
}

:::::::::::::: psh1.c ::::::::::::::

#include	<stdio.h>
#include	<signal.h>

/**	prompting shell version 1
 **
 **		Prompts for the command and its arguments.
 **		Builds the argument vector for the call to execvp.
 **		Uses execvp(), and never returns.
 **/

#define	MAXARGS		20				/* cmdline args		*/
#define	ARGLEN		100				/* token length		*/
#define	CMDPROMPT	"Command please: "		/* prompts	*/
#define	ARGPROMPT	"next argument please: "
#define	prompt(n)	printf("%s", (n)==0 ? CMDPROMPT : ARGPROMPT );

main()
{
	char	*arglist[MAXARGS+1];		/* an array of ptrs	*/
	int	numargs;			/* index into array	*/
	char	argbuf[ARGLEN];				/* read stuff here	*/
	char	*makestring();			/* malloc etc		*/

	numargs = 0;
	while ( numargs < MAXARGS )
	{
		prompt( numargs );	/* prompt user for arg */

					/* if arg, add to list */
		if ( gets(argbuf) && *argbuf ) {
			arglist[numargs++] = makestring(argbuf);
		}
		else
		{
					/* done, exit or execute cmd	*/
			if ( numargs == 0 )		/* no command? */
				break;			/* y: get out	*/
			arglist[numargs] = NULL ;	/* close list	*/
			execute( arglist );		/* do it	*/
			numargs = 0;			/* and reset	*/
		}
	}
	return 0;
}
```

```
execute( char *arglist[] )
/*
 *      use execvp to do it
 */
{
        execvp(arglist[0], arglist);                    /* do it */
        perror("execvp failed");
        exit(1);
}

char *
makestring( char *buf )
{
        char    *cp, *malloc();
        if ( cp = malloc( strlen(buf) + 1 ) ){                  /* get mem      */
                strcpy(cp, buf);                        /* copy chars  */
                return cp;                              /* return ptr  */
        }
        fprintf(stderr,"out of memory\n");
        exit(1);
}

::::::::::::: forkdemo.c :::::::::::::
#include        <stdio.h>

/**
 **     forkdemo1.c
 **
 **             shows how fork creates two processes, distinguishable
 **             by the different return values from fork()
 **/

main()
{
        int     ret_from_fork, my_pid;

        my_pid = getpid();                      /* who am i?           */
        printf("Hi, my pid is %d\n", my_pid );         /* tell the world      */

        ret_from_fork = fork();

        sleep(1);
        printf("after fork().  It returned %d and my pid is %d\n",
                        ret_from_fork, getpid());
}
```

```
:::::::::::::: forkdemo2.c ::::::::::::::

#include	<stdio.h>

main()
{
	int	what_fork_said;

	printf("I am about to fork...\n");

	what_fork_said = fork();

	if ( what_fork_said == 0 )
	{
		printf("I am the child process.  I am process id %d\n", getpid());
	}
	else if ( what_fork_said != -1 )
	{
		printf("I am the parent, my child is %d\n", what_fork_said);
	}
	else
		perror("fork");
}
:::::::::::::: waitdemo.c ::::::::::::::

#include	<stdio.h>

/*
 *	 shows how wait() returns the pid of an exiting child
 *
 */

main()
{
	int	ret_from_fork,
		ret_from_wait;

	char	*args[4];
	int	child_status;


	printf(" about to fork....\n");

	ret_from_fork = fork();

	if ( ret_from_fork == -1 ){
		perror( "fork" );
		exit(1);
	}

	if ( ret_from_fork == 0 ){		/* the child	*/
		args[0] = "ps";
		args[1] = "x";
		args[2] = NULL ;
		printf("I am the child and will do ps\n");

		execvp( "ps", args );		/* do the command	*/
		perror( "execvp" );
		exit(1);
	}

	printf("Parent here, sleeping until child exits..\n");

	ret_from_wait = wait( &child_status );

	printf("wait returned a value of %d\n", ret_from_wait );
	printf("and the child process exited with code %d\n", child_status);
}
```

```
:::::::::::::: psh2.c ::::::::::::::

#include       <stdio.h>
#include       <signal.h>

/**
 **    prompting shell version 2
 **
 **            Solves the 'one-shot' problem of version 1
 **                    Uses execvp(), but fork()s first so that the
 **                    shell waits around to perform another command
 **            New problem: shell catches signals.  Run vi, press ^c.
 **/

#define MAXARGS        20                              /* cmdline args        */
#define ARGLEN         100                             /* token length        */
#define CMDPROMPT      "Command please: "              /* prompts     */
#define ARGPROMPT      "next argument please: "
#define prompt(n)      printf("%s", (n)==0 ? CMDPROMPT : ARGPROMPT );

main()
{
        char    *arglist[MAXARGS+1];           /* an array of ptrs    */
        int     numargs;                       /* index into array    */
        char    argbuf[ARGLEN];                        /* read stuff here     */
        char    *makestring();                 /* malloc etc          */

        numargs = 0;
        while ( numargs < MAXARGS )
        {
                prompt( numargs );     /* prompt user for arg */

                                       /* if arg, add to list */
                if ( gets(argbuf) && *argbuf ) {
                        arglist[numargs++] = makestring(argbuf);
                }
                else
                {
                                       /* done, exit or execute cmd  */
                        if ( numargs == 0 )            /* no command? */
                                break;                 /* y: get out  */
                        arglist[numargs] = NULL ;      /* close list  */
                        execute( arglist );            /* do it       */
                        numargs = 0;                   /* and reset   */
                }
        }
        return 0;
}

execute( char *arglist[] )
/*
 *      use fork and execvp and wait to do it
 */
{
        int     pid,exitstatus;                                /* of child    */

        pid = fork();                                  /* make new process */
        switch( pid ){
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        execvp(arglist[0], arglist);           /* do it */
                        perror("execvp failed");
                        exit(1);
                default:
                        while( wait(&exitstatus) != pid )
                                ;
                        printf("child exited with status %d,%d\n",
                                       exitstatus>>8, exitstatus&0377);
        }
}
char *
makestring( char *buf )
{
  /* same as in psh1.c */
}
```