Recursive Structures and I/O

COM S 113

February 10, 1999

Announcements

Read K&R Chapter 6

`sizeof` Operator

Expressions "`sizeof` *expression*" and "`sizeof(`*type name*`)`" yield integral size in bytes

If given an expression, *the expression is not evaluated*

Type of `sizeof` expression is `size_t`, defined in `<stddef.h>`

Cannot be applied to functions, bit fields, or objects of incomplete type

`sizeof` Operator (continued)

```
int main() {
  char a[100];
  int b[50];

  printf("sizeof a[14] is %d\n", sizeof a[14]);
  printf("sizeof a is %d\n", sizeof a);
  printf("sizeof b[14] is %d\n", sizeof b[14]);
  printf("b has %d elements\n", sizeof b / sizeof b[0]);
}
```


Pointers to Structures

Expensive to pass large structures between functions

Instead pass pointer to structure:

```
void printpoint(struct point *pp) {
  printf("(%d,%d)", (*pp).x, (*pp).y);        }
int main() { struct point origin = { 0, 0 };
  printpoint(&origin);                        }
```


Pointers to Structures (continued)

Operator `->` is shorthand for accessing a field given a structure pointer

```
void printpoint(struct point *pp) {
  printf("(%d,%d)", pp->x, pp->y);
      /* same as (*pp).x and (*pp).y */
}
```


Pointers to Structures (continued)

Operators `->` and `.` associate left-to-right, have maximum precedence along with `()` and `[]`

For example, `++pp->x` increments field `x`, not the pointer `pp`

Memory Allocation Functions

All four declared in `<stdlib.h>`

`void *malloc(size_t size)` allocates `size` bytes and returns a pointer to the new space if possible; otherwise it returns null pointer

`void *calloc(size_t n, size_t size)` same as `malloc(n * size)` except that allocated storage is zeroed

Memory Allocation Functions (continued)

`void *realloc(void *ptr, size_t size)` changes size of previously allocated object to `size` and returns pointer to new space if possible; otherwise it returns null pointer

`void free(void *ptr)` deallocates previously allocated storage

Self-Referential Structures

```
#include <stdlib.h>

typedef struct node {
  int datum;
  struct node *next;
} node;
```



```
node *newlist(void) {
  node *t = malloc(sizeof(node));
  if (t == NULL) fatal("newlist: out of storage");
  t->next = NULL;
  return t;
}

void freelist(node *list) {
  if (list->next != NULL) freelist(list->next);
  free(list);
}
```

```
int empty(node *list) {
  return list->next == NULL;
}

int in(node *list, int d) {
  node *t;

  for (t = list->next; t != NULL; t = t->next)
    if (t->datum == d) return 1;
  return 0;
}
```



```
void insert(node *list, int d) {
  if (!in(list, d)) {
    node *t = malloc(sizeof(node));
    if (t == NULL)
      fatal("insert: out of storage");
    t->datum = d;
    t->next = list->next;
    list->next = t;
  }
}
```



```
void delete(node *list, int d) {
  node *t;
  for (t = list;
       t->next != NULL && t->next->datum != d;
       t = t->next) /* null statement */ ;
  if (t->next != NULL) {
    node *del = t->next;
    t->next = del->next;
    free(del);
  }
}
```


Example Use of List

```
int main() {
  node *list = newlist();
  insert(list, 4); insert(list, 8); insert(list, 12);
  delete(list, 4); delete(list, 7); delete(list, 12);
  printf("The list is %sempty\n",
         empty(list) ? "" : "not ");
  freelist(list);
}
```

Input and Output Streams

All I/O is done through "streams"; two kinds: *text* and *binary*

Text streams are sequences of lines, each of which is a sequence of characters terminated by a newline

Binary streams are sequences of characters corresponding to the internal representation of data

Streams (continued)

Streams are created by opening files

Streams are referenced using stream pointers (of type `FILE *`, defined in `<stdio.h>`

Normally three *standard streams* are automatically opened: `stdin`, `stdout`, and `stderr`

Stream Functions

All stream functions described in K&R section B1

Whenever a function takes a stream as a parameter, the stream is the first parameter

Using Output Streams

`putc`	Write a character to the specified stream (macro)
`fputc`	Same as `putc` (but a function)
`putchar`	Write a character to `stdout`
`puts`	Write a string to `stdout`
`fputs`	Write a string to the specified stream
`printf`	Write the list of values to `stdout` according to the format string
`fprintf`	Write the list of values to the specified stream according to the format string

Example: Writing to Streams

```
#include <stdio.h>
#include <stdlib.h>

void fatal(char *s) {
  fprintf(stderr, "Error, %s\n", s);
  exit(EXIT_FAILURE); /* defined in <stdlib.h> */
}
```


Using Input Streams

`getc`	Get next char from specified stream (macro)
`fgetc`	Same as `getc` (but a function)
`getchar`	Get next char from `stdin`
`scanf`	Read values from `stdin` according to format string
`fscanf`	Read values from specified stream according to format string
`gets`	Get a string from `stdin`
`fgets`	Get a string from specified stream

Warning: Never Use `gets`!

`char *gets(char *s)` reads from `stdin` until newline, replacing newline with `'\0'`

`char *fgets(char *s, int n, FILE *stream)` reads until newline *or until* $n-1$ *characters read*, appending `'\0'`

Danger with `gets()`: possible for user to overrun array bounds

Accessing Files

Open files with `fopen()`, close them with `fclose()`

`FILE *fopen(const char *fname, const char *mode)`

Note: `mode` is a *string*, not a character!

Text File Modes

`"r"`	Open text file for reading
`"w"`	Create text file for writing (truncates)
`"a"`	Open or create text file for appending
`"r+"`	Open text file for update (read and write)
`"w+"`	Create text file for update (truncates)
`"a+"`	Open or create text file for update & appending

Most commands to read from text streams return `EOF` (defined in `<stdio.h>`) on error condition

File Access Example

Suppose our employee database file has lines like the following:

```
mharris 5162 5-7421
fleming 5162 5-7421
liz     4126 5-8593
```


We could define a structure like this:

```
#include <stdio.h>
#include <stdlib.h>

struct {
  char name[20], room[5], phone[7];
} db[100];
```



```
int main() {
  FILE *fp;
  char *dbfile = "database.txt";
  int i = 0;

  if ((fp = fopen(dbfile, "r")) == NULL) {
    fprintf(stderr, "Can't open %s\n", dbfile);
    exit(EXIT_FAILURE);
  }
```

```
  while (fscanf(fp, "%s%s%s",
                db[i].name, db[i].room,
                db[i].phone) == 8) {
    printf("Read record %d for name '%s'\n",
           i, db[i].name);
    i++;
  }
  fclose(fp);
}
```