Fall 2006 Handout 10

The tree of data types in C

You can augment the declaration

```
int x;                  /* the simplest possible declaration */
```

in three ways: add a **`*`** to make a ‘‘pointer to’’, or a **`[10]`** to make an ‘‘array of’’, or a **`(void)`** to make a ‘‘function returning’’.

```
int *x;                 /* the 1st way: pointer to int */
int x[10];              /* the 2nd way: array of int's */
int x(void);            /* the 3rd way: function returning int */
```

Similarly, you can augment the declaration

```
int *x;                 /* pointer to int */
```

in the same three ways:

```
int **x;                /* the 1st way: pointer to pointer to int */
int *x[10];             /* the 2nd way: array of pointers to int */
int *x(void);           /* the 3rd way: function returning pointer to int */
```

The following tree begins with the most simple declaration, and progresses to its increasingly complicated descendants.

Not every declaration can be augmented in all three ways; some can be augmented in only the first way or the first two ways. For example, a declaration for a function can be augmented in only the first way, e.g., the only child of line 20 is line 21. And a declaration for an array can be augmented in only the first two ways, e.g., the only children of line 12 are lines 13 and 17. But every other declaration can be augmented in all three ways, e.g. the three children of line 1 are lines 2, 12, and 20.

What is the rational for this syntax? Suppose **`x`** is a ‘‘pointer to pointer to pointer to **`int`**’’ (line 4) which has been given a legal value. What operators and parentheses would you have to apply to **`x`** to print the value of the **`int`** that is ultimately pointed to?

```
printf("%d\n", ***x);
```

Similarly, suppose **`x`** is an ‘‘array of pointer to pointer to **`int`**’’ (line 5) which has been filled with legal values. To print the value of one of the **`int`**’s that are ultimately pointed to,

```
printf("%d\n", **x[10]);     /* assume the subscripts go up to [10] */
```

In every case, the punctuation in the above **`printf`**’s is the same as the punctuation in the declarations below.


```
 1 int x;                          int
 2    int *x;                      pointer to int
 3       int **x;                  pointer to pointer to int
 4          int ***x;              pointer to pointer to pointer to int
 5          int **x[10];           array of pointer to pointer to int
 6          int **x(void);         function returning pointer to pointer to int
 7       int *x[10];               array of pointer to int
 8          int *(*x)[10];         pointer to array of pointer to int
 9          int *x[10][10];        array of array of pointer to int
10       int *x(void);             function returning pointer to int
11          int *(*x)(void);       pointer to function returning pointer to int
12    int x[10];                   array of int
13       int (*x)[10];             pointer to array of int
14          int (**x)[10];         pointer to pointer to array of int
15          int (*x[10])[10];      array of pointer to array of int
16          int (*x(void))[10];    function returning pointer to array of int
17       int x[10][10];            array of array of int
18          int (*x)[10][10];      pointer to array of array of int
19          int x[10][10][10];     array of array of array of int
20    int x(void);                 function returning int
21       int (*x)(void);           pointer to function returning int
22          int (**x)(void);       pointer to pointer to function returning int
23          int (*x[10])(void);    array of pointer to function returning int
24          int (*x(void))(void);  function returning pointer to function returning int
```

Definition. *Array of array of* means ‘‘two-dimensional array’’ in the above diagram (lines 17, 9, 18); and similarly for three-dimensional arrays (line 19).

Definition. An *evil twin* of a variable is another variable whose declaration differs only by the parentheses for grouping (not the parentheses around an argument list), e.g.,

```
int *x[10];             /* line 7 */
int (*x)[10];           /* line 13 */

int *x(void);           /* line 10 */
int (*x)(void);         /* line 21 */

int **x[10];            /* line 5: evil triplets */
int *(*x)[10];          /* line 8 */
int (**x)[10];          /* line 14 */
```

Unusual storage classes

```
1 int i;
2 const int i = 10;       /* K&R pp. 40, 211; King pp. 407−408: */
3                         /* must give initial value and no other */
4 register int i;         /* K&R pp. 83-84, 210; King pp. 405−406: */
5                         /* only inside of a function */
6 volatile int i;         /* K&R p. 211; King p. 464: */
7                         /* hardware may change variable’s value */
```

Build a linked list: K&R pp. 139−143; King pp. 370−383

Each structure occupies its own individually **`malloc`**’ed block. Each will contain a field giving the address of the next structure in the list.

A structure cannot contain another copy of the same kind of structure (this would cause an infinite regress), but it can contain a pointer to another copy of the same kind of structure. Lines 17−20 show how to do this.

```
 1 /* Excerpt from /usr/include/stdio.h: K&R p. 102; King p. 391
 2 #define NULL 0
```

```
 1 /* Excerpt from /usr/include/stddef.h: K&R p. 103; King p. 471
 2 typedef long unsigned int size_t;
```

```
 1 /* Let the user type in numbers and store them in a dynamically allocated linked
 2 list.  Each incoming number is stored at the head of the list, so the newest
 3 number is at the head and the oldest number is at the tail.  Then print the list
 4 from head to tail.
 5
 6 Each number is stored in a structure called a "node".  Each node is malloc'ed
 7 separately.  The variable "head" holds the address of the first node, or is NULL
 8 if there are no nodes yet.
 9
10 The "next" field of each node holds the address of the next node, or is NULL if
11 there is no next node.  The 0 that the user types in to terminate the input is
12 not stored in the list. */
13
14 #include <stdio.h>
15 #include <stdlib.h>
16
17 typedef struct node {         /* self-referential: K&R p. 140; King pp. 370−371 */
18     int n;                    /* the number */
19     struct node *next;        /* need the word "struct" in self-reference */
20 } node;
21
22 void *mymalloc(size_t n);     /* Handout 8, pp. 7-8 */
23
24 main()
25 {
26     node *head = NULL;
27     node *p;
28     int n;
29
30     printf("Please type numbers.\n");
31     printf("Press RETURN after each number.\n");
32     printf("Type 0 when done.\n");
33
34     for (;;) {
35         scanf("%d", &n);
36         if (n <= 0) {
37             break;
38         }
39
40         p = (node *)mymalloc(sizeof(node));
41         p->n = n;
42         p->next = head;
43         head = p;
```

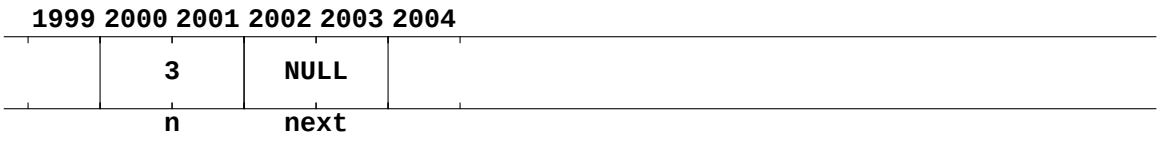
```
44      }
45
46      /* Print the list from head to tail. */
47      printf("head == %p\n", p);
48      for (p = head; p != NULL; p = p->next) {
49          printf("p == %p, p->n == %d, p->next == %p\n", p, p->n, p->next);
50      }
51
52      exit(EXIT_SUCCESS);
53  }
```

Page 38 shows that lines 30−32 can be combined to

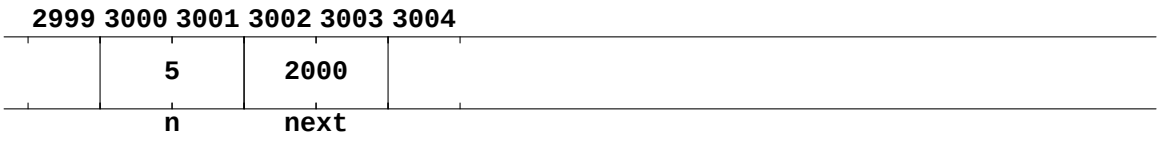
```
30      printf("Please type numbers.\n"
31          "Press RETURN after each number.\n"
32          "Type 0 when done.\n");
```

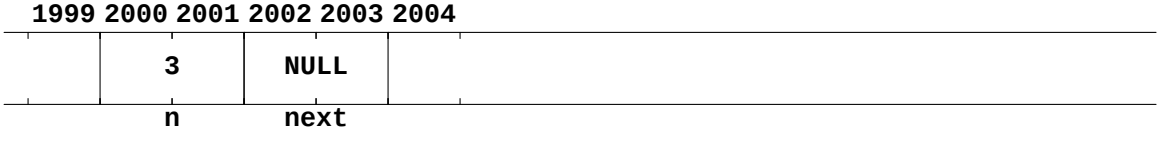
```
Please type numbers.
Press RETURN after each number.
Type 0 when done.
3
5
1
0
head == 1000
p == 1000, p->n == 1, p->next == 3000
p == 3000, p->n == 5, p->next == 2000
p == 2000, p->n == 3, p->next == 0
```

The first call to **mymalloc** allocated the four bytes 2000−2003. The first time we finish line 43, the value of **head** is 2000 and the linked list looks like this:

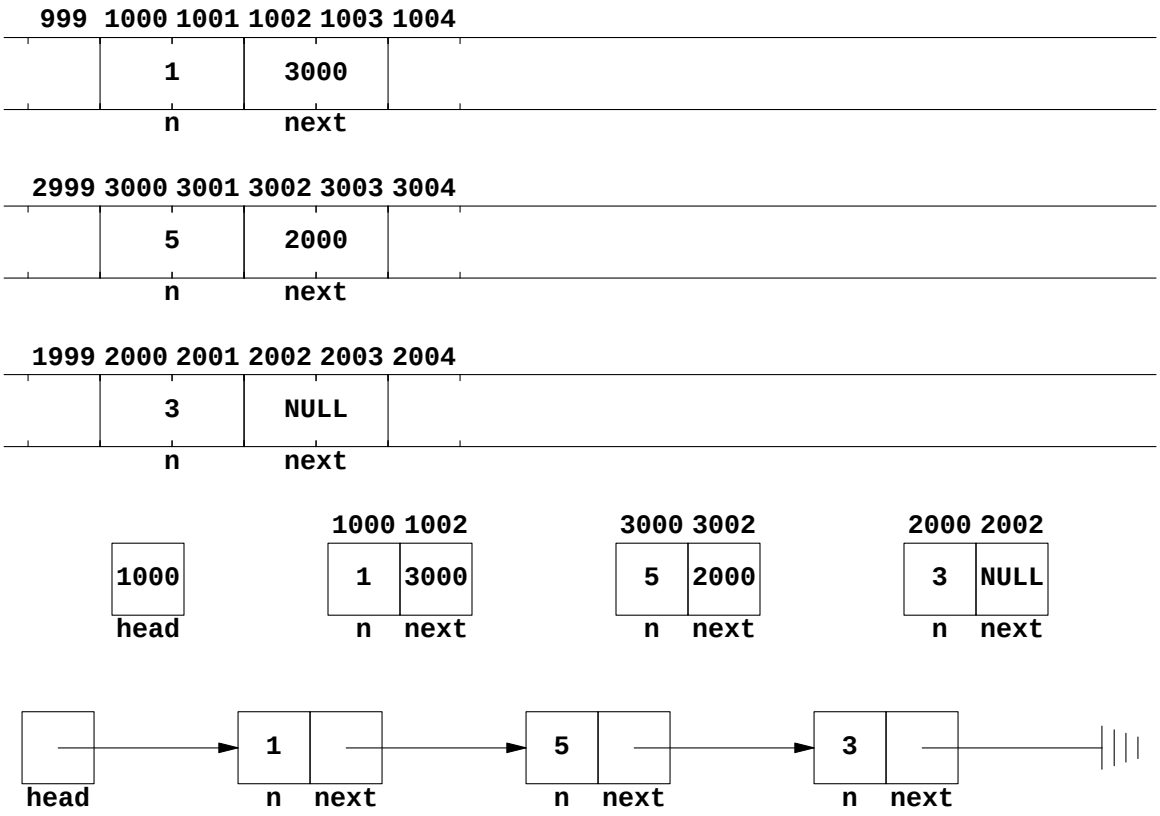


The second call to **mymalloc** allocated the four bytes 3000−3003. The second time we finish line 43, the value of **head** is 3000 and the linked list looks like this:





The third call to **mymalloc** allocated the four bytes 1000−1003. The third time we finish line 43, the value of **head** is 1000 and the linked list looks like this:



Append the numbers to the tail of the list

```
 1 /* Let the user type in numbers and store them in a dynamically allocated linked
 2 list.  Each incoming number is stored at the tail of the list, so the newest
 3 number is at the tail and the oldest number is at the head.  Then print the list
 4 from head to tail.
 5
 6 Each number is stored in a structure called a "node".  Each node is malloc'ed
 7 separately.  The variable "head" holds the address of the first node, or is NULL
 8 if there are no nodes yet.  The variable "tail" holds the address of the last
 9 node, or is NULL if there are no nodes yet.
10
11 The "next" field of each node holds the address of the next node, or is NULL if
12 there is no next node.  The 0 that the user types in to terminate the input is
13 not stored in the list. */
14
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 typedef struct node {         /* self-referential: K&R p. 140; King pp. 370-371 */
19     int n;                    /* the number */
20     struct node *next;        /* need the word "struct" in self-reference */
21 } node;
22
23 void *mymalloc(size_t n);     /* Handout 8, pp. 6-7 */
24
25 main()
26 {
27     node *head = NULL;
```

```
28      node *tail = NULL;
29      node *p;
30      int n;
31
32      printf("Please type numbers.\n"
33          "Press RETURN after each number.\n"
34          "Type 0 when done.\n");
35
36      for (;;) {
37          scanf("%d", &n);
38          if (n <= 0) {
39              break;
40          }
41
42          p = (node *)mymalloc(sizeof(node));
43          p->n = n;
44          p->next = NULL;
45
46          if (head == NULL) {
47              head = p;
48          }
49
50          if (tail != NULL) {
51              tail->next = p;
52          }
53          tail = p;
54      }
55
56      /* Print the list from head to tail. */
57      for (p = head; p != NULL; p = p->next) {
58          printf("%d\n", p->n);
59      }
60
61      exit(EXIT_SUCCESS);
62  }
```

```
Please type numbers.
Press RETURN after each number.
Type 0 when done.
3
5
1
0
3
5
1
```

Delete nodes from a singly linked list: K&R pp. 139−143; King pp. 376−378

```
1  /* Let the user type in positive numbers and store them in a dynamically
2  allocated linked list.  Each incoming number is stored at the head of the list,
3  so the newest number is at the head and the oldest number is at the tail.
4
5  If the user types a negative number, remove the corresponding positive number
```



```
 6 from the list or output an error message if it's not in the list.  When the user
 7 types a zero, print the list from head to tail.
 8
 9 Each number is stored in a structure called a "node".  Each node is malloc'ed
10 separately.  The variable "head" holds the address of the first node, or is NULL
11 if there are no nodes yet.
12
13 The "next" field of each node holds the address of the next node, or is NULL if
14 there is no next node.  The 0 that the user types in to terminate the input is
15 not stored in the list. */
16
17 #include <stdio.h>
18 #include <stdlib.h>
19
20 typedef struct node {         /* self-referential: K&R p. 140; King pp. 370–371 */
21     int n;                    /* the number */
22     struct node *next;        /* need the word "struct" in self-reference */
23 } node;
24
25 void *mymalloc(size_t n);     /* Handout 8, pp. 6-7 */
26
27 main()
28 {
29     node *head = NULL;
30     node *p, *q;
31     int n;
32
33     printf("Please type numbers.\n");
34     printf("Press RETURN after each number.\n");
35     printf("Type 0 when done.\n");
36
37     for (;;) {
38         scanf("%d", &n);
39         if (n == 0) {
40             break;
41         } else if (n > 0) {
42             /* Insert a new number into the list. */
43             p = malloc(sizeof(node));
44             p->n = n;
45             p->next = head;
46             head = p;
47         } else if (head == NULL) {
48             printf("%d was not in the list.\n", n);
49         } else if (head->n == -n) {
50             /* Let q point to the node to be removed. */
51             q = head;
52             head = q->next;
53             free (q);              /* Handout 8, p. 5, line 24 */.
54         } else {
55             for (p = head;; p = p->next) {
56                 q = p->next;
57                 if (q == NULL) {
58                     printf("%d was not in the list.\n", n);
59                     break;
```

```
60                  }
61
62                  if (q->n == -n) {
63                      /* q points to the node to be removed. */
64                      p->next = q->next;
65                      free (q);
66                      break;
67                  }
68              }
69          }
70      }
71
72      /* Print the list from head to tail.*/
73      printf("head == %p\n", p);
74      for (p = head; p != NULL; p = p->next) {
75          printf("p->n == %d\n", p->n);
76      }
77
78      exit(EXIT_SUCCESS);
79  }
```

Lines 56−57 may be combined to

```
56                  if ((q = p->next) == NULL) {
```

just like

```
    while ((c = getchar()) != EOF) {
```

```
Please type numbers.
Press RETURN after each number.
Type 0 when done.
1
5
3
-5
0
3
1
```

Build a sorted linked list: K&R pp. 139−143; King pp. 378−383

There are three cases in the following program:

(1) Insert the first node into a previously empty list (lines 44−48).

(2) Insert a new node at the start of the list (lines 50−54).

(3) Insert a new node after an existing node (lines 56−68).

```
1 /* Let the user type in positive numbers, not necessarily unique.  Store them
2 in ascending order in a dynamically allocated linked list.  The smallest number
3 will be at the head of the list, and the biggest number at the tail.  Then print
4 the list from head to tail.
5
6 Each number is stored in a structure called a "node".  Each node is malloc'ed
7 separately.  The variable "head" holds the address of the first node, or is NULL
8 if there are no nodes yet.
```



```
 9
10 The "next" field of each node holds the address of the next node, or is NULL if
11 there is no next node.  The 0 that the user types in to terminate the input is
12 not stored in the list. */
13
14 #include <stdio.h>
15 #include <stdlib.h>
16
17 typedef struct node {
18     int n;  /* the number */
19     struct node *next;
20 } node;
21
22 void *mymalloc(size_t n);     /* Handout 8, pp. 6-7 */
23
24 main()
25 {
26     node *head = NULL;        /* the address of the first node on the list */
27     node *new;                /* the new node that the user typed in */
28     node *p;                  /* for looping through the list */
29     int n;                    /* each number that the user types in */
30
31     printf("Please type positive numbers.\n");
32     printf("Press RETURN after each number.\n");
33     printf("Type 0 when done.\n");
34
35     for (;;) {
36         scanf("%d", &n);
37         if (n <= 0) {
38             break;
39         }
40
41         new = (node *)mymalloc(sizeof(node));
42         new->n = n;
43
44         if (head == NULL) {
45             /* The list was empty. */
46             new->next = NULL;
47             head = new;
48         }
49
50         else if (new->n <= head->n) {
51             /* Insert new at the head of the list. */
52             new->next = head;
53             head = new;
54         }
55
56         else {
57             /* Search the list to find the insertion point. */
58             for (p = head; p->next != NULL; p = p->next) {
59                 if (p->next->n >= new->n) {
60                     break;
61                 }
62             }
```

```
63
64              /* Insert new immediately after the node that p points to. */
65              new->next = p->next;
66              p->next = new;
67          }
68      }
69
70      /* Print the list from head to tail. */
71      for (p = head; p != NULL; p = p->next) {
72          printf("%d\n", p->n);
73      }
74
75      exit(EXIT_SUCCESS);
76  }
```

```
Please type positive numbers.
Press RETURN after each number.
Type 0 when done.
1
5
3
0
1
3
5
```

In line 46, **head** is always equal to **NULL** because of line 44. Therefore you can change **NULL** to **head** in line 46. Now consolidate lines 46−47 to lines 52−53 using **||** as in Handout 5, p. 6.

A doubly linked list: K&R pp. 139−143; King pp. 378−383

We can loop through the above list in only one direction, because all the pointers point one way. To loop through the list in either direction, we can make a doubly linked list instead:

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/doubly.c

```
 1 /*
 2 Let the user type in positive numbers, not necessarily unique.  Store them in
 3 ascending order in a dynamically allocated doubly linked list.  The smallest
 4 number will be at the head of the list, and the biggest number at the tail.
 5 Then print the list from head to tail and from tail to head.
 6
 7 Each number is stored in a structure called a node_t.  Each node is malloc'ed
 8 separately.  head is the first node, tail is always the last.
 9
10 The next field of each node holds the address of the next node, or is NULL if
11 there is no next node.  The prev field of each node holds the address of the
12 previous node, or is NULL if there is no previous node.  The 0 that the user
13 types in to terminate the input is not stored in the list.
14 */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <limits.h>  /* for INT_MAX */
19
```

```
20 typedef struct node_t {
21     int n;                  /* the payload */
22     struct node_t *prev;    /* the address of the previous node */
23     struct node_t *next;    /* the address of the next node */
24 } node_t;
25
26 int main()
27 {
28     node_t head = {      0, NULL, NULL};
29     node_t tail = {INT_MAX, NULL, NULL};
30
31     node_t *p, *q;   /* for looping through the list */
32     int n;           /* each number that the user types in */
33
34     head.next = &tail;
35     tail.prev = &head;
36
37     printf(
38         "Please type positive numbers less than %d.\n"
39         "Press RETURN after each number.\n"
40         "Type a negative number to delete the corresponding positive number.\n"
41         "Type 0 when done.\n", INT_MAX);
42
43     for (;;) {
44         scanf("%d", &n);
45         if (n == 0) {
46             break;
47         }
48
49         if (n > 0) { /* Insert a new node. */
50             node_t *new = malloc(sizeof(node_t));
51             if (new == NULL) {
52                 fprintf(stderr, "Can't allocate %lu bytes.\n", sizeof(node_t));
53                 return EXIT_FAILURE;
54             }
55             new->n = n;
56
57             /*
58             Search the list to find the insertion point.
59             The insertion point will always be between head and tail.
60             */
61             for (p = &head; (q = p->next) != &tail; p = q) {
62                 if (q->n >= n) {
63                     break;
64                 }
65             }
66
67             /* Insert the new node between the ones that p and q point to. */
68             printf("Found the insertion point.\n");
69             new->next = q;
70             new->prev = p;
71             p->next = new;
72             q->prev = new;
73         }
```

```
 74
 75          else {  /* Delete an existing node. */
 76              n = -n;
 77
 78              /* Let p point to the node to be deleted. */
 79              for (p = head.next; p != &tail; p = p->next) {
 80                  if (p->n == n) {
 81                      goto found;
 82                  }
 83              }
 84              fprintf(stderr, "The number %d is not on the list.\n", n);
 85              continue;
 86
 87              found:;
 88              p->prev->next = p->next;
 89              p->next->prev = p->prev;
 90              free(p);
 91          }
 92      }
 93
 94      printf("Print the list from head to tail:\n");
 95      for (p = head.next; p != &tail; p = p->next) {
 96          printf("%d\n", p->n);
 97      }
 98
 99      printf("\nPrint the list from tail to head:\n");
100      for (p = tail.prev; p != &head; p = p->prev) {
101          printf("%d\n", p->n);
102      }
103
104      return EXIT_SUCCESS;
105  }
```

▼ Homework 10.1: build (or dismantle) a linked data structure

Write a program that **`malloc`**'s many structures, all of the same **`typedef`**. Each structure should be linked to one or two of the others, i.e., each structure should contain the address of one or two of the others. Describe how the data structure is linked together: ''The bleep field holds the address of the next bloop, or is **`NULL`** if there is no next bloop.'' ''The variable blip holds the address of the first blop, or is **`NULL`** if there aren't any blops yet.''

Do one of the following:

(1) Make a doubly linked list of **`node`**'s. The field named **`next_input`** will link the **`node`**'s in the order in which they were input (most recent at the head). The field named **`next_size`** will link the **`node`**'s in increasing size order (smallest at the head). Make a variable named **`head_input`** to hold the address of the last **`node`** that was input, or **`NULL`** if no **`node`** has been input yet. Make a variable named **`head_size`** to hold the address of the smallest **`node`** that was input, or **`NULL`** if no **`node`** has been input yet. Do not make any **`tail`** variables. When the user types a 0, print the list from head to tail in both orders.

(2) Take the above singly linked list of sorted numbers and allow the user to remove as well as insert numbers. If the user types 3, insert the number 3. If he or she types −3, delete the number 3 if it is in the list, or print an error message if not. After unlinking the node from the list, give the address of the node to **`free`**. Do not make any **`tail`** variable. When the user types a 0, print the list from head to tail.

(3) After you learn about recursion, rewrite the tree in K&R pp. 139−143 to hold **`int`**'s in ascending order instead of strings in alphabetical order.

▲

A gentle introduction to recursion: K&R pp. 86−88; King pp. 172−176

This program prints the integers from 1 to 10 without a loop. Moreover, all the variables are constants: we never apply a **++** or an **=** to any variable.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/looper.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void looper(int n);
 5
 6 int main()
 7 {
 8     looper(1);           /* Start printing at 1. */
 9     return EXIT_SUCCESS;
10 }
11
12 /* Print the integers from n to 10 inclusive. */
13
14 #if 0
15 void looper(int n)
16 {
17     int i;
18
19     for (i = n; i <= 10; ++i) {
20         printf("%d\n", i);
21     }
22 }
23 #endif
24
25 void looper(int n)
26 {
27     if (n <= 10) {
28         printf("%d\n", n);
29         looper(n + 1);
30     }
31 }
```

```
1
2
3
4
5
6
7
8
9
10
```

▼ **Homework 10.2: parameterize the vital statistics**

A recursive function can do anything that a **`for`** loop can do.

(1) Make the function **`looper`** start printing at 2 instead of at 1.

(2) Make it count by twos instead of by ones: 2, 4, 6, 8, 10.

(3) Make it print forever: 2, 4, 6, 8, 10, 12, ... Don't add anything to the function **`looper`**: you have to remove something. Then put it back.

(4) Instead of hard-wiring the number 10 into the function **`looper`**, pass it to **`looper`** as a second argument. When **`looper`** calls itself in line 29, just pass along the second argument unchanged. Have **`main`** call **`looper`** to print the numbers from 1 to 20:

```
 1 int main()
 2 {
 3     looper(1, 20);      /* Now looper takes two arguments. */
```

Also pass the increment as a third argument:

```
 4     looper(1, 20, 1);
```

In C++, let the default value of the last argument be 1.

(5) [Difficult question.] What happens if you swap lines 28 and 29? Why?

▲

Process each character in a string using recursion

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/lower.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <ctype.h>   /* for tolower: KR pp. 166, 248-249; King p. 528 */
 4
 5 #define N 256    /* maximum word length */
 6 void lower(char *p);
 7
 8 int main()
 9 {
10     char a[N];
11
12     printf("Please type a word and press RETURN.\n");
13     scanf("%s", a);
14     lower(a);
15     printf("The word in all lowercase is %s.\n", a);
16
17     return EXIT_SUCCESS;
18 }
19
20 /* Change the string to all lowercase. */
21
22 #if 0
23 void lower(char *string)
24 {
25     char *p;
26
27     for (p = string; *p != '\0'; ++p) {
28         *p = tolower(*p);
29     }
```

```
30 }
31 #endif
32
33 void lower(char *p)
34 {
35     if (*p != '\0') {
36         *p = tolower(*p);
37         lower(p + 1);
38     }
39 }
```

```
Please type a word and press RETURN.
CompuServe
The word in all lowercase is compuserve.
```

A recursive function that returns a value: compute a factorial

"*n* factorial", written *n*!, is the product of all the integers from 1 to *n.* For example,

$$4! = 1 \times 2 \times 3 \times 4 = 24.$$

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/factorial.c

```
 1 /* Let the user type in a number n.  Then output n!. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 long factorial(long n);
 6
 7 int main()
 8 {
 9     long n;
10
11     printf("Please type in a number and press RETURN.\n");
12     scanf("%ld", &n);
13
14     printf("%ld\n", factorial(n));
15     return EXIT_SUCCESS;
16 }
17
18 /* Return n!. */
19
20 #if 0
21 long factorial(long n)
22 {
23     long i;
24     long product = 1;
25
26     for (i = 1; i <= n ++i) {        /* Why not start at zero? */
27         product *= i;
28     }
29     return product;
30 }
31 #endif
```

```
32
33 #if 0
34 long factorial(long n)
35 {
36     if (n <= 1) {
37         return 1;
38     }
39
40     return n * factorial(n - 1);
41 }
42 #endif
43
44 long factorial(long n)
45 {
46     return n <= 1 ? 1 : n * factorial(n - 1);
47 }
```

```
Please type in a number and press RETURN.
4
24
```

A recursive function that returns the greatest common divisor of two integers

```
1 int gcd(int i, int j)
2 {
3     return j == 0 ? i : gcd(j, i % j);
4 }
```

A simpler version of the recursive function on K&R p. 87

The following recursive method is simpler than the non-recursive method in Handout 7, pp. 13−14:

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/printd.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>    /* for log10 and pow */
 4
 5 void printd(int n);
 6
 7 int main()
 8 {
 9     printd(12345);
10     printf("\n");
11     return EXIT_SUCCESS;
12 }
13
14 /* Print a non-negative int in decimal. */
15
16 #if 0
17 void printd(int n)
18 {
19     int i;
20
```



```
21      for (i = pow(10, (int)log10(n)); i >= 1; i /= 10) {
22          printf("%d", n / i % 10);
23      }
24  }
25  #endif
26
27  void printd(int n)
28  {
29      /* If n has any digits in addition to the rightmost digit, */
30      if (n >= 10) {
31          printd(n / 10);     /* Print all but the rightmost digit of n. */
32      }
33
34      printf("%d", n % 10);   /* Print the rightmost digit of n. */
35  }
```

```
12345
```

Instead of writing the number 10 over and over, create a

```
const int base = 10;
```

Also change the expression **`log10(n)`** to **`(log(x) / log(10))`**, and then change the **`10`** to **`base`**.

Print the linked list in Handout 10, pp. 3–4, without a loop

Add the following function definition to the bottom of the program. Remember to add the corresponding function declaration to the top of the program. Cf. **`treeprint`**, K&R p. 142.

```
1  /* Print the numbers in the linked list whose first node is p, starting at p. */
2
3  void listprint(node *p)
4  {
5      if (p != NULL) {
6          printf("p == %p, p->n == %d, p->next == %p\n", p, p->n, p->next);
7          listprint(p->next);
8      }
9  }
```

Then replace the **`for`** loop in lines 49–51 with the statement

```
listprint(head);
```

Why eliminate the `for` loops?

`for` loops are adequate for repeating the same task over and over again, e.g., for printing every item in a list or in a two-dimensional array. Unfortunately, non-linear and non-rectangular data structures such as trees can not be printed by **`for`** loops, or even by nested **`for`** loops.

To process these self-nested structures we need another plan of action: recursion. I wrote the recursive **`listprint`** function to help you read the slightly more complicated **`treeprint`** function on K&R p. 142. Although **`listprint`** can be written more simply with a **`for`** loop, there is no way to write **`treeprint`** with a **`for`** loop.

```
1  /* Print the numbers in the binary tree whose first node is p, starting at p. */
2
3  void treeprint(node *p)
4  {
5      if (p != NULL) {
```



```
 6          treeprint(p->left);
 7          printf("p->n == %d\n", p->n);
 8          treeprint(p->right);
 9      }
10  }
```

To get into the spirit of recursion, look up the last of the six page numbers listed in the Index of the textbook under ‘‘recursion’’.

How to write a recursive program

(1) Write the computation as a separate function which will call itself.

(2) Have the new function do only the first step of the job. For example, **looper** prints only the first of a series of numbers, **lower** changes only the first **char** of a string, **printd** prints only the rightmost digit of a number, **factorial** does only the first of a series of multiplications, and **listprint** prints only the first structure on the linked list.

(3) Then have the function call itself to do the rest of the job. Shave off the part of the job that has already been done, and pass the remainder as an argument in the call to itself. For example, **looper** passes **n+1** to itself instead of all of **n**, **printd** passes **n/10** to itself instead of all of **n**, **lower** passes **p+1** to itself instead of all of **p**, **factorial** passes **n-1** to itself instead of all of **n**, and **listprint** passes **p->next** to itself instead of all of **p**.

Sometimes step (2) is done before step (3): see **looper**, **lower**, **factorial**, and **listprint**. Sometimes step (3) is done before step (2): see **printd**. Sometimes step (3) is split into two halves and part (2) is done in the middle; see the example on top of K&R p. 142. and the famous ‘‘Towers of Hanoi’’ problem.

(4) The call to itself *must* be inside of an **if** statement, or else the recursion would never end. The **if** statement prevents the function from calling itself again when the job is already finished.

Find a path through a maze with recursion

From pp. 71−77 of the greatest book about programming: *The Elements of Programming Style, second edition* by Brian W. Kernighan and P. J. Plauger; McGraw-Hill, 1978; ISBN 0-07-034207-5.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/maze.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int try(int row, int col);
 5
 6 #define MAXCOL 17
 7 char maze[][MAXCOL] = {
 8     "XXXXXXXXXXXXXXXXX",
 9     "X   XXXXXXXXXXXX",
10     "X XXXXXXXXXXXXXX",
11     "X XXXXXXXXXXXXXX",
12     "X   XXXXXXXXXXXX",
13     "X X XXXXXXXXXXXX",
14     "X X XXXXXXXXXXXX",
15     "XXX XXXXXXX    XX",
16     "XXX XXXXXXX XX XX",
17     "XXX         XX XX",
18     "XXX XXX XXX XXXXX",
19     "X   XXX XXX X   X",
20     "X XXXXX XXX X X X",
```

```
21      "X   XXX XXX XfX X",
22      "XXX XXX XXXXXXX X",
23      "XXX             X",
24      "XXXXXXXXXXXXXXXXX"
25  };
26  #define MAXROW (sizeof maze / sizeof maze[0])
27
28  int main()
29  {
30      int row;
31      int col;
32
33      if (!try(1, 1)) {
34          printf("No path starting at 1, 1.\n");
35          return EXIT_FAILURE;
36      }
37
38      for (row = 0; row < MAXROW; ++row) {
39          for (col = 0; col < MAXCOL; ++col) {
40              printf("%c", maze[row][col]);
41          }
42          printf("\n");
43      }
44
45      return EXIT_SUCCESS;
46  }
47
48  /*
49  If there is a path from this location to the finish, draw the path with plus
50  signs.  Return 1 if there is a path, 0 otherwise.
51  */
52
53  int try(int row, int col)
54  {
55      /* If we're off the board, */
56      if (row < 0 || row >= MAXROW || col < 0 || col >= MAXCOL) {
57          return 0;
58      }
59
60      /* If we're already at the "finish", */
61      if (maze[row][col] == 'f') {
62          return 1;
63      }
64
65      /* If this location is already occupied by an 'X' or '+', */
66      if (maze[row][col] != ' ') {
67          return 0;
68      }
69
70      /* Arrive here if we're at an empty location on the board.
71      Optimistically assume that it's the first step of a path leading to the
72      finish. */
73      maze[row][col] = '+';
74
```



```
75      if (try(row + 1, col) ||        /* down */
76          try(row - 1, col) ||        /* up */
77          try(row, col + 1) ||        /* left */
78          try(row, col - 1)) {        /* right */
79          return 1;
80      }
81
82      /* Our earlier optimism proved to be unfounded. */
83      maze[row][col] = ' ';
84
85      return 0;
86  }
```

```
XXXXXXXXXXXXXXXXX
X+  XXXXXXXXXXXXX
X+XXXXXXXXXXXXXXX
X+XXXXXXXXXXXXXXX
X+++XXXXXXXXXXXXX
X X+XXXXXXXXXXXXX
X X+XXXXXXXXXXXXX
XXX+XXXXXXX    XX
XXX+XXXXXXX XX XX
XXX+        XX XX
XXX+XXX XXX XXXXX
X+++XXX XXX X+++X
X+XXXXX XXX X+X+X
X+++XXX XXX XfX+X
XXX+XXX XXXXXXX+X
XXX++++++++++++X
XXXXXXXXXXXXXXXXX
```

Given a choice, the above snake prefers to move down because ‘‘down’’ is tried first in lines 75−78.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/shortest.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <limits.h>  /* for INT_MAX */
 4
 5 int try(int row, int col, int draw);
 6
 7 #define MAXCOL 17
 8 char maze[][MAXCOL] = {
 9     "XXXXXXXXXXXXXXXXX",
10     "X   XXXXXXXXXXXX",
11     "X XXXXXXXXXXXXXX",
12     "X XXXXXXXXXXXXXX",
13     "X   XXXXXXXXXXXX",
14     "X X XXXXXXXXXXXX",
15     "X X XXXXXXXXXXXX",
16     "XXX XXXXXXX    XX",
17     "XXX XXXXXXX XX XX",
18     "XXX         XX XX",
19     "XXX XXX XXX XXXXX",
20     "X   XXX XXX X   X",
```

```
21      "X XXXXX XXX X X X",
22      "X   XXX XXX XfX X",
23      "XXX XXX XXXXXXX X",
24      "XXX             X",
25      "XXXXXXXXXXXXXXXXX"
26  };
27  #define MAXROW (sizeof maze / sizeof maze[0])
28
29  int main()
30  {
31      int row;
32      int col;
33      const int length =  try(1, 1, 1);
34
35      if (length == INT_MAX) {
36          printf("No path starting at 1, 1.\n");
37          return EXIT_FAILURE;
38      }
39
40      printf("A shortest path is of length %d.\n\n", length);
41
42      for (row = 0; row < MAXROW; ++row) {
43          for (col = 0; col < MAXCOL; ++col) {
44              printf("%c", maze[row][col]);
45          }
46          printf("\n");
47      }
48
49      return EXIT_SUCCESS;
50  }
51
52  /*
53  Return the length of the shortest path from this location to the finish.
54  If there is no path, return INT_MAX.
55  Draw the path if the third argument is non-zero.
56  */
57
58  int try(int row, int col, int draw)
59  {
60      typedef struct {
61          int drow;
62          int dcol;
63      } direction_t;
64
65      static const direction_t a[] = {
66          {-1, 0},   /* down */
67          { 1, 0},   /* up */
68          { 0,-1},   /* left */
69          { 0, 1}    /* right */
70      };
71  #define MAXDIRECTION (sizeof a / sizeof a[0])
72
73      const direction_t *p;
74      const direction_t *direction;
```

```
 75     int min = INT_MAX;
 76
 77     /* If we're off the board, */
 78     if (row < 0 || row >= MAXROW || col < 0 || col >= MAXCOL) {
 79         return INT_MAX;
 80     }
 81
 82     /* If we're already at the "finish", */
 83     if (maze[row][col] == 'f') {
 84         return 0;
 85     }
 86
 87     /* If this location is already occupied by an 'X' or '+', */
 88     if (maze[row][col] != ' ') {
 89         return INT_MAX;
 90     }
 91
 92     /* Arrive here if we're at an empty location on the board.
 93     Optimistically assume that it's the first step of a path leading to the
 94     finish. */
 95     maze[row][col] = '+';
 96
 97     for (p = a; p < a + MAXDIRECTION; ++p) {
 98         const int length = try(row + p->drow, col + p->dcol, 0);
 99         if (length < min) {
100             direction = p;
101             min = length;
102         }
103     }
104
105     /* Our earlier optimism proved to be unfounded. */
106     if (min == INT_MAX) {
107         maze[row][col] = ' ';
108         return INT_MAX;
109     }
110
111     if (draw) {
112         /* Now that we've identified the first step from here along the
113         shortest path, draw the rest of the path. */
114         try(row + direction->drow, col + direction->dcol, draw);
115     } else {
116         /* If we're not supposed to draw, erase what we drew. */
117         maze[row][col] = ' ';
118     }
119
120     return min + 1;
121 }
```



```
XXXXXXXXXXXXXXXXX
X+  XXXXXXXXXXXXX
X+XXXXXXXXXXXXXXX
X+XXXXXXXXXXXXXXX
X+++XXXXXXXXXXXXX
X X+XXXXXXXXXXXXX
X X+XXXXXXXXXXXXX
XXX+XXXXXXX    XX
XXX+XXXXXXX XX XX
XXX+++++    XX XX
XXX XXX+XXX XXXXX
X   XXX+XXX X+++X
X XXXXX+XXX X+X+X
X   XXX+XXX XfX+X
XXX XXX+XXXXXXX+X
XXX    +++++++++X
XXXXXXXXXXXXXXXXX
```

Sort an array with recursion (invented in 1960)

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/quicksort.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4 #define N 10 /* number of int's to sort. */
 5
 6 void quicksort(int *begin, int *end);
 7 void swap(int *p1, int *p2);
 8
 9 int main()
10 {
11     int a[N];
12     int i;
13
14     srand(time(NULL));
15     for (i = 0; i < N; ++i) {
16         a[i] = rand();
17     }
18
19     quicksort(a, a + N);
20
21     for (i = 0; i < N; ++i) {
22         printf("%d\n", a[i]);
23     }
24
25     return EXIT_SUCCESS;
26 }
27
28 /*
29 Sort the int's into increasing order.  begin is the address of the first int;
30 end is one more than the address of the last int.  The number of int's to sort
31 is therefore (end - begin).
32 */
```



```
33
34 void quicksort(int *begin, int *end)
35 {
36     int *p;
37     int *last;
38     const ptrdiff_t length = end - begin;
39
40     if (length <= 1) {
41         return;
42     }
43
44     /* Pick a random element and move it to the front. */
45     swap(begin, begin + rand() % length);
46
47     /* Move all the elements that are smaller than the randomly selected
48     element to the range begin + 1 to last inclusive. */
49
50     /* address of the last element that is < the randomly selected element. */
51     last = begin;
52
53     for (p = begin + 1; p < end; ++p) {
54         if (*p < *begin) {
55             swap(++last, p);
56         }
57     }
58
59     /* Put the randomly selected element where it belongs. */
60     swap(begin, last);
61
62     /* Sort all the int's that are < the randomly selected element. */
63     quicksort(begin, last);
64
65     /* Sort all the int's that are >= the randomly selected element. */
66     quicksort(last + 1, end);
67 }
68
69 void swap(int *p1, int *p2)
70 {
71     const int temp = *p1;
72     *p1 = *p2;
73     *p2 = temp;
74 }

      1162
      3728
      4980
      4984
      5406
      7355
      9797
      16503
      26763
      26806
```

How deeply does **quicksort** get called? The first time we call it, it sorts **N** numbers. The first time it calls itself, it sorts **N/2** numbers. When that call calls itself, it sorts **N/4** numbers. **quicksort** therefore goes $\log_2$ **N** levels down. For example, it goes 4 levels down to sort 16 numbers.

How many times do we execute the comparison in line 54? **N** times at the first level, **N** times at the second level, **N** times at every level, for a total of **N** $\log_2$ **N** times.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/bubble.c

```
 1 /* Bubble sort an array of 10 ints into ascending order.  The for loop in lines
 2 29-38 will move the array elements part of the way into the correct order.
 3
 4 If some moves were made, it means that we should execute this for loop again to
 5 see if additional moves will be made.  In this case, flag is set to 1 to make
 6 the do-while loop execute the for loop again.
 7
 8 If no moves were made, it means that the elements are already in order.  In this
 9 case, flag remains 0 and the do-while loop terminates. */
10
11 #include <stdio.h>
12
13 main()
14 {
15     int a[10];
16     int i;       /* index into the array */
17     int flag;    /* set to 1 to ensure one more trip */
18     int temp;    /* temporary storage for exchanging values */
19
20     /* Initialize the array with the numbers to be sorted. */
21     printf("Type %d numbers.  Press RETURN after each one.\n", 10);
22     for (i = 0; i < 10; ++i) {
23         scanf("%d", &a[i]);
24     }
25
26     /* Bubble sort the array into ascending order. */
27     do {
28         flag = 0;
29         for (i = 0; i < 9; ++i) {
30             if (a[i] > a[i+1]) {
31                 temp = a[i];     /* swap a[i] and a[i+1] */
32                 a[i] = a[i+1];
33                 a[i+1] = temp;
34
35                 flag = 1;
36             }
37             printf("debug: i == %d\n", i);
38         }
39     } while (flag == 1);
40
41     /* Output the array. */
42     for (i = 0; i < 10; ++i) {
43         printf("%11d\n", a[i]);
44     }
45 }
```


How many times do we execute the comparison in line 30?

How not to program in any language

```
 1 int BigFieldJustChanged(char *);    /* return 0 for false, non-zero for true */
 2
 3     char CELL[3][3];                /* tic-tac-toe board */
 4     char WINNER[2];                 /* an empty string */
 5
 6     if (BigFieldJustChanged(&CELL[0][0])) {
 7         if (CELL[0][0] == 'X' && CELL[0][1] == 'X' && CELL[0][2] == 'X' ||
 8             CELL[0][0] == 'X' && CELL[1][0] == 'X' && CELL[2][0] == 'X' ||
 9             CELL[0][0] == 'X' && CELL[1][1] == 'X' && CELL[2][2] == 'X') {
10             strcpy(WINNER, "X");
11         }
12
13         if (CELL[0][0] == 'O' && CELL[0][1] == 'O' && CELL[0][2] == 'O' ||
14             CELL[0][0] == 'O' && CELL[1][0] == 'O' && CELL[2][0] == 'O' ||
15             CELL[0][0] == 'O' && CELL[1][1] == 'O' && CELL[2][2] == 'O') {
16             strcpy(WINNER, "O");
17         }
18     }
19
20     if (BigFieldJustChanged(&CELL[1][0])) {
21         if (CELL[1][0] == 'X' && CELL[1][1] == 'X' && CELL[1][2] == 'X' ||
22             CELL[0][0] == 'X' && CELL[1][0] == 'X' && CELL[2][0] == 'X') {
23             strcpy(WINNER, "X");
24         }
25
26         if (CELL[1][0] == 'O' && CELL[1][1] == 'O' && CELL[1][2] == 'O' ||
27             CELL[0][0] == 'O' && CELL[1][0] == 'O' && CELL[2][0] == 'O') {
28             strcpy(WINNER, "O");
29         }
30     }
31
32     if (BigFieldJustChanged(&CELL[2][0])) {
33         if (CELL[2][0] == 'X' && CELL[2][1] == 'X' && CELL[2][2] == 'X' ||
34             CELL[2][0] == 'X' && CELL[1][0] == 'X' && CELL[0][0] == 'X' ||
35             CELL[2][0] == 'X' && CELL[1][1] == 'X' && CELL[0][2] == 'X') {
36             strcpy(WINNER, "X");
37         }
38
39         if (CELL[2][0] == 'O' && CELL[2][1] == 'O' && CELL[2][2] == 'O' ||
40             CELL[2][0] == 'O' && CELL[1][0] == 'O' && CELL[0][0] == 'O' ||
41             CELL[2][0] == 'O' && CELL[1][1] == 'O' && CELL[0][2] == 'O') {
42             strcpy(WINNER, "O");
43         }
44     }
45
46     if (BigFieldJustChanged(&CELL[0][1])) {
47         if (CELL[0][1] == 'X' && CELL[1][1] == 'X' && CELL[2][1] == 'X' ||
48             CELL[0][1] == 'X' && CELL[0][0] == 'X' && CELL[0][2] == 'X') {
49             strcpy(WINNER, "X");
50         }
51
```

```
52          if (CELL[0][1] == 'O' && CELL[1][1] == 'O' && CELL[2][1] == 'O' ||
53              CELL[0][1] == 'O' && CELL[0][0] == 'O' && CELL[0][2] == 'O') {
54              strcpy(WINNER, "O");
55          }
56      }
57
58      if (BigFieldJustChanged(&CELL[1][1])) {
59          if (CELL[1][1] == 'X' && CELL[2][1] == 'X' && CELL[0][1] == 'X' ||
60              CELL[1][1] == 'X' && CELL[1][0] == 'X' && CELL[1][2] == 'X' ||
61              CELL[1][1] == 'X' && CELL[0][0] == 'X' && CELL[2][2] == 'X' ||
62              CELL[1][1] == 'X' && CELL[0][2] == 'X' && CELL[2][0] == 'X') {
63              strcpy(WINNER, "X");
64          }
65
66          if (CELL[1][1] == 'O' && CELL[2][1] == 'O' && CELL[0][1] == 'O' ||
67              CELL[1][1] == 'O' && CELL[1][0] == 'O' && CELL[1][2] == 'O' ||
68              CELL[1][1] == 'O' && CELL[0][0] == 'O' && CELL[2][2] == 'O' ||
69              CELL[1][1] == 'O' && CELL[0][2] == 'O' && CELL[2][0] == 'O') {
70              strcpy(WINNER, "O");
71          }
72      }
73
74      if (BigFieldJustChanged(&CELL[2][1])) {
75          if (CELL[2][1] == 'X' && CELL[2][0] == 'X' && CELL[2][2] == 'X' ||
76              CELL[2][1] == 'X' && CELL[0][1] == 'X' && CELL[1][1] == 'X') {
77              strcpy(WINNER, "X");
78          }
79
80          if (CELL[2][1] == 'O' && CELL[2][0] == 'O' && CELL[2][2] == 'O' ||
81              CELL[2][1] == 'O' && CELL[0][1] == 'O' && CELL[1][1] == 'O') {
82              strcpy(WINNER, "O");
83          }
84      }
85
86      if (BigFieldJustChanged(&CELL[0][2])) {
87          if (CELL[0][2] == 'X' && CELL[1][2] == 'X' && CELL[2][2] == 'X' ||
88              CELL[0][2] == 'X' && CELL[0][1] == 'X' && CELL[0][0] == 'X' ||
89              CELL[0][2] == 'X' && CELL[1][1] == 'X' && CELL[2][0] == 'X') {
90              strcpy(WINNER, "X");
91          }
92
93          if (CELL[0][2] == 'O' && CELL[1][2] == 'O' && CELL[2][2] == 'O' ||
94              CELL[0][2] == 'O' && CELL[0][1] == 'O' && CELL[0][0] == 'O' ||
95              CELL[0][2] == 'O' && CELL[1][1] == 'O' && CELL[2][0] == 'O') {
96              strcpy(WINNER, "O");
97          }
98      }
99
100     if (BigFieldJustChanged(&CELL[1][2])) {
101         if (CELL[1][2] == 'X' && CELL[0][2] == 'X' && CELL[2][2] == 'X' ||
102             CELL[1][2] == 'X' && CELL[1][1] == 'X' && CELL[1][0] == 'X') {
103             strcpy(WINNER, "X");
104         }
105
```

```
106         if (CELL[1][2] == 'O' && CELL[0][2] == 'O' && CELL[2][2] == 'O' ||
107             CELL[1][2] == 'O' && CELL[1][1] == 'O' && CELL[1][0] == 'O') {
108             strcpy(WINNER, "O");
109         }
110     }
111
112     if (BigFieldJustChanged(&CELL[2][2])) {
113         if (CELL[2][2] == 'X' && CELL[1][2] == 'X' && CELL[0][2] == 'X' ||
114             CELL[2][2] == 'X' && CELL[2][1] == 'X' && CELL[2][0] == 'X' ||
115             CELL[2][2] == 'X' && CELL[1][1] == 'X' && CELL[0][0] == 'X') {
116             strcpy(WINNER, "X");
117         }
118
119         if (CELL[2][2] == 'O' && CELL[1][2] == 'O' && CELL[0][2] == 'O' ||
120             CELL[2][2] == 'O' && CELL[2][1] == 'O' && CELL[2][0] == 'O' ||
121             CELL[2][2] == 'O' && CELL[1][1] == 'O' && CELL[0][0] == 'O') {
122             strcpy(WINNER, "O");
123         }
124     }
```

Invent two new data types: cell_t and line_t

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define N 3             /* number of rows == number of columns */
 5 char CELL[N][N];         /* tic-tac-toe board, inherited from previous program. */
 6
 7 /* Each cell has a row number and a column number. */
 8 typedef struct {
 9     int row;
10     int col;
11 } cell_t;
12
13 /* A line is N cells in a straight line. */
14 typedef cell_t line_t[N];
15
16 /* Here are the lines on a tic-tac-toe board: */
17
18 const line_t line[] = {
19     {{0, 0}, {0, 1}, {0, 2}},       /* top row */
20     {{1, 0}, {1, 1}, {1, 2}},       /* middle row */
21     {{2, 0}, {2, 1}, {2, 2}},       /* bottom row */
22
23     {{0, 0}, {1, 0}, {2, 0}},       /* left column */
24     {{0, 1}, {1, 1}, {2, 1}},       /* middle column */
25     {{0, 2}, {1, 2}, {2, 2}},       /* right column */
26
27     {{0, 0}, {1, 1}, {2, 2}},       /* main diagonal (upper left to lower right) */
28     {{0, 2}, {1, 1}, {2, 0}}        /* other diagonal (upper right to lower left) */
29 };
30 #define  NLINES (sizeof line / sizeof line[0])
31
32 int BigFieldJustChanged(char *cell);  /* inherited from previous program */
```



```
33
34 cell_t which_cell_just_changed();
35 int contains(const line_t line, cell_t cell);
36 char all_the_same(const line_t line);
37
38 int main(int argc, char **argv)
39 {
40     int i;  /* index into line[] array (better name than l) */
41     char c;
42     char WINNER[] = "?"; /* inherited from previous program */
43
44
45     cell_t cell = which_cell_just_changed();
46     if (cell.row == -1 || cell.col == -1) {
47         fprintf(stderr, %s: no cell changed.\n", argv[0]);
48         return EXIT_FAILURE;
49     }
50
51     for (i = 0; i < NLINES; ++i) {
52         if (contains(line[i], cell) && (c = all_the_same(line[i])) != '\0') {
53             WINNER[0] = c;
54             printf("The winner is %s\n", WINNER);
55             return EXIT_SUCCESS;
56         }
57     }
58
59     return EXIT_FAILURE;
60 }
61
62 /*
63 Return the cell_t that just changed.  This function does the work of the widely
64 scattered lines 6, 20, 32, 46, 58, 74, 89, 100, 112 of the original program.
65 */
66
67 cell_t which_cell_just_changed()
68 {
69     cell_t not_found = {-1, -1};
70
71     for (row = 0; row < N; ++row) {
72         for (col = 0; col < N; ++col) {
73             if (BigFieldJustChanged(&CELL[row][col])) {
74                 cell_t cell = {row, col};
75                 return cell;
76             }
77         }
78     }
79
80     return not_found;
81 }
82
83 /*
84 If the line contains the specified cell, return 1.  Otherwise, return 0.
85 */
86
```



```
 87 int contains(const line_t line, cell_t cell)
 88 {
 89     int i;
 90
 91     for (i = 0; i < N; ++i) {
 92         if (line[i].row == cell.row && line[i].col == cell.col) {
 93             return 1;
 94         }
 95     }
 96
 97     return 0;
 98 }
 99
100 /*
101 If all N characters in the line are the same, return the character.
102 Otherwise, return '\0'.
103 */
104
105 char all_the_same(const line_t line)
106 {
107     const char c = CELL[line[0].row][line[0].col];    /* first char in the line */
108     int i;
109
110     /* all the remaining char's */
111     for (i = 1; i < N; ++i) {
112         if (c != CELL[line[i].row][line[i].col]) {
113             return '\0';
114         }
115     }
116
117     return c;
118 }
```


Do it with a Perl regular expression

http://i5.nyu.edu/~mm64/x52.9544/src/winner

```
#!/bin/perl

$_ = <STDIN>;
chomp($_);  #Remove the trailing newline from $_.

if (length($_) != 9 || $_ =~ /[^XO ]/) {
    die "$0: Input line must be nine X's, O's, or blanks.";
}

#Insert a dash after the 3rd character and after the 6th character.
#For example, OOXXXOOXX becomes OOX-XXO-OXX.

$_ =~ s/(...)(...)(...)/\1-\2-\3/;

if (
    $_ =~ /([XO])\1\1/   ||      #any row
    $_ =~ /([XO])...\1...\1/ ||     #any column
    $_ =~ /([XO])..-.\1.-..\1/ ||   #the main diagonal
    $_ =~ /([XO])-.\1.-\1/) {       #the other diagonal

    print "$1 is a winner.\n";
    exit 0;
}

print "No one has won yet.\n";
exit 1;
```

□

