Fall 2004 Handout 6

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

—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9232/src/looper.c**

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

void looper(int n);

int main()
{
    looper(1);          /* Start printing at 1. */
    return EXIT_SUCCESS;
}

/* Print the integers from n to 10 inclusive. */

#if 0
void looper(int n)
{
    int i;

    for (i = n; i <= 10; ++i) {
        printf("%d\n", i);
    }
}
#endif

void looper(int n)
{
    if (n <= 10) {
        printf("%d\n", n);
        looper(n + 1);
    }
}
```

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

▼ Homework 6.1: 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:

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

Also pass the increment as a third argument:

```
    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

—Source code on the Web at **`http://i5.nyu.edu/˜mm64/x52.9232/src/lower.c`**

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>   /* for tolower: KR pp. 166, 248-249; King p. 528 */

#define N 256    /* maximum word length */
void lower(char *p);

int main()
{
    char a[N];

    printf("Please type a word and press RETURN.\n");
    scanf("%s", a);
    lower(a);
    printf("The word in all lowercase is %s.\n", a);

    return EXIT_SUCCESS;
}
```

```

/* Change the string to all lowercase. */

#if 0
void lower(char *string)
{
    char *p;

    for (p = string; *p != '\0'; ++p) {
        *p = tolower(*p);
    }
}
#endif

void lower(char *p)
{
    if (*p != '\0') {
        *p = tolower(*p);
        lower(p + 1);
    }
}
```

```
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,

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

—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9232/src/factorial.c**

```
/* Let the user type in a number n.  Then output n!. */
#include <stdio.h>
#include <stdlib.h>

long factorial(long n);

int main()
{
    long n;

    printf("Please type in a number and press RETURN.\n");
    scanf("%ld", &n);

    printf("%ld\n", factorial(n));
    return EXIT_SUCCESS;
}

/* Return n!. */

#if 0
long factorial(long n)
```

```
{
    long i;
    long product = 1;

    for (i = 1; i <= n ++i) {        /* Why not start at zero? */
        product *= i;
    }
    return product;
}
#endif

#if 0
long factorial(long n)
{
    if (n <= 1) {
        return 1;
    }

    return n * factorial(n - 1);
}
#endif

long factorial(long n)
{
    return n <= 1 ? 1 : n * factorial(n - 1);
}
```

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

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

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

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:

—Source code on the Web at `http://i5.nyu.edu/˜mm64/x52.9232/src/printd.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>    /* for log10 and pow */

void printd(int n);

int main()
{
    printd(12345);
    printf("\n");
    return EXIT_SUCCESS;
}

/* Print a non-negative int in decimal. */

#if 0
void printd(int n)
{
    int i;

    for (i = pow(10, (int)log10(n)); i >= 1; i /= 10) {
        printf("%d", n / i % 10);
    }
}
#endif

void printd(int n)
{
    /* If n has any digits in addition to the rightmost digit, */
    if (n >= 10) {
        printd(n / 10);     /* Print all but the rightmost digit of n. */
    }

    printf("%d", n % 10);   /* Print the rightmost digit of n. */
}
```

```
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 6, 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.

```
/* Print the numbers in the linked list whose first node is p, starting at p. */

void listprint(node *p)
{
    if (p != NULL) {
        printf("p == %p, p->n == %d, p->next == %p\n", p, p->n, p->next);
        listprint(p->next);
    }
}
```

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.

```
/* Print the numbers in the binary tree whose first node is p, starting at p. */

void treeprint(node *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("p->n == %d\n", p->n);
        treeprint(p->right);
    }
}
```

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.

—Source code on the Web at **`http://i5.nyu.edu/˜mm64/x52.9232/src/maze.c`**

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

int try(int row, int col);

#define MAXCOL 17
char maze[][MAXCOL] = {
    "XXXXXXXXXXXXXXXX",
    "X   XXXXXXXXXXXX",
    "X XXXXXXXXXXXXXX",
    "X XXXXXXXXXXXXXX",
    "X   XXXXXXXXXXXX",
    "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"
};
#define MAXROW (sizeof maze / sizeof maze[0])

int main()
{
    int row;
    int col;

    if (!try(1, 1)) {
        printf("No path starting at 1, 1.\n");
        return EXIT_FAILURE;
    }

    for (row = 0; row < MAXROW; ++row) {
        for (col = 0; col < MAXCOL; ++col) {
            printf("%c", maze[row][col]);
        }
        printf("\n");
    }

    return EXIT_SUCCESS;
}

/*
If there is a path from this location to the finish, draw the path with plus
signs.  Return 1 if there is a path, 0 otherwise.
*/

int try(int row, int col)
{
    /* If we’re off the board, */
    if (row < 0 || row >= MAXROW || col < 0 || col >= MAXCOL) {
        return 0;
    }

    /* If we’re already at the "finish", */
    if (maze[row][col] == ’f’) {
        return 1;
    }

    /* If this location is already occupied by an ’X’ or ’+’, */
    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.

—Source code 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   XXXXXXXXXXXXX",
11     "X XXXXXXXXXXXXXXX",
12     "X XXXXXXXXXXXXXXX",
13     "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"
};
#define MAXROW (sizeof maze / sizeof maze[0])

int main()
{
    int row;
    int col;
    const int length =  try(1, 1, 1);

    if (length == INT_MAX) {
        printf("No path starting at 1, 1.\n");
        return EXIT_FAILURE;
    }

    printf("A shortest path is of length %d.\n\n", length);

    for (row = 0; row < MAXROW; ++row) {
        for (col = 0; col < MAXCOL; ++col) {
            printf("%c", maze[row][col]);
        }
        printf("\n");
    }

    return EXIT_SUCCESS;
}

/*
Return the length of the shortest path from this location to the finish.
If there is no path, return INT_MAX.
Draw the path if the third argument is non-zero.
*/

int try(int row, int col, int draw)
{
    typedef struct {
        int drow;
        int dcol;
    } direction_t;

    static const direction_t a[] = {
        {-1, 0},   /* down */
        { 1, 0},   /* up */
        { 0,-1},   /* left */
        { 0, 1}    /* right */
    };
#define MAXDIRECTION (sizeof a / sizeof a[0])

    const direction_t *p;
    const direction_t *direction;
    int min = INT_MAX;

    /* If we’re off the board, */
    if (row < 0 || row >= MAXROW || col < 0 || col >= MAXCOL) {
        return INT_MAX;
    }

    /* If we’re already at the "finish", */
    if (maze[row][col] == ’f’) {
        return 0;
    }

    /* If this location is already occupied by an ’X’ or ’+’, */
    if (maze[row][col] != ’ ’) {
        return INT_MAX;
    }

    /* Arrive here if we’re at an empty location on the board.
    Optimistically assume that it’s the first step of a path leading to the
    finish. */
    maze[row][col] = ’+’;

    for (p = a; p < a + MAXDIRECTION; ++p) {
        const int length = try(row + p->drow, col + p->dcol, 0);
        if (length < min) {
            direction = p;
            min = length;
        }
    }

    /* Our earlier optimism proved to be unfounded. */
    if (min == INT_MAX) {
        maze[row][col] = ’ ’;
        return INT_MAX;
    }

    if (draw) {
        /* Now that we’ve identified the first step from here along the
        shortest path, draw the rest of the path. */
        try(row + direction->drow, col + direction->dcol, draw);
    } else {
        /* If we’re not supposed to draw, erase what we drew. */
        maze[row][col] = ’ ’;
    }

    return min + 1;
}
```

```
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)

—Source code on the Web at **`http://i5.nyu.edu/˜mm64/x52.9232/src/quicksort.c`**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 10 /* number of int’s to sort. */

void quicksort(int *begin, int *end);
void swap(int *p1, int *p2);

int main()
{
    int a[N];
    int i;

    srand(time(NULL));
    for (i = 0; i < N; ++i) {
        a[i] = rand();
    }

    quicksort(a, a + N);

    for (i = 0; i < N; ++i) {
        printf("%d\n", a[i]);
    }

    return EXIT_SUCCESS;
}

/*
Sort the int’s into increasing order.  begin is the address of the first int;
end is one more than the address of the last int.  The number of int’s to sort
is therefore (end - begin).
*/

```

```
void quicksort(int *begin, int *end)
{
    int *p;
    int *last;
    const ptrdiff_t length = end - begin;

    if (length <= 1) {
        return;
    }

    /* Pick a random element and move it to the front. */
    swap(begin, begin + rand() % length);

    /* Move all the elements that are smaller than the randomly selected
    element to the range begin + 1 to last inclusive. */

    /* address of the last element that is < the randomly selected element. */
    last = begin;

    for (p = begin + 1; p < end; ++p) {
        if (*p < *begin) {
            swap(++last, p);
        }
    }

    /* Put the randomly selected element where it belongs. */
    swap(begin, last);

    /* Sort all the int’s that are < the randomly selected element. */
    quicksort(begin, last);

    /* Sort all the int’s that are >= the randomly selected element. */
    quicksort(last + 1, end);
}

void swap(int *p1, int *p2)
{
    const int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

```
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.

—Source code 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?

□