C Supplementary

Terry Marris August 2012

Recursion

We look at function calls, iterations or loops, and recursive function calls. A recursive function call involves a function that calls itself.

Function Calls

A function call says: *hey, function, go and do your job with the data values I give you, then give me your result*.

```
/* sum.c - uses a function to add two integers */

#include <stdio.h>

int sum(int a, int b)
{
  return a + b;
}


int main()
{
  int result = sum(2, 3);
  printf("Result is %d\n", result);
  return 0;
}
```

function call

Program run

```
Result is 5
```

The function is named *sum()*. It has two integer parameters named *a* and *b*. The function returns an integer, the result of adding *a* to *b*.

In the function call *sum(2, 3)*, *2* is passed to *a*, *3* is passed to *b*, and the value returned by the function *sum()*, namely *5*, is assigned to *result*.

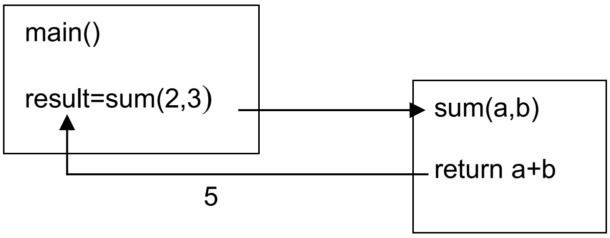


Fig 1 *sum(2,3)* calls *sum(a,b)*. *sum(a,b)* returns *5*

Iteration

An iteration, or loop, is used to execute a block of statements again and again.

```
/* goloop.c - uses a loop to print Go Go Go */

#include <stdio.h>

int printGo(int ntimes)
{
  int i = ntimes;
  while (i > 0) {
    printf("Go ");
    i--;
  }
  printf("\n");
  return 0;
}

int main()
{
  printGo(3);
  return 0;
}
```

Program run

```
Go Go Go
```

The loop is

```
int i = ntimes;
while (i > 0) {
  printf("Go ");
  i--;
}
```

int i = *ntimes* is the initialisation stage. Whatever is stored in *ntimes* e.g. 3, is assigned to the integer variable *i*. The initialisation stage is executed just the once.

while introduces the continuation condition. The continuation condition is *(i* > *0)*. If *i* is greater than zero then loop.

Each time round the loop *Go* is printed and the value of *i* is decreased by 1.

So, if *i* starts off with the value 3, and each time round the loop the value of *i* decreases, there must come a time when *i* is no longer more than zero. *i not* > *0* is the stopping condition. *i not* > *0* is the same as *i* <= *0*. So the stopping condition, when the loop is no longer executed, is *i* <= *0*;

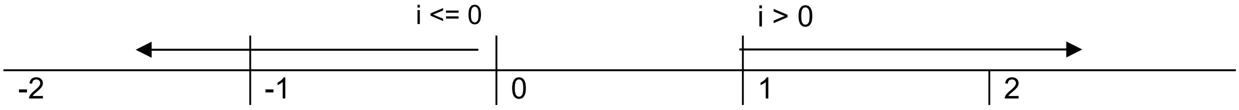


Fig 2 Number line showing the numbers that are less or equal to 0, and the numbers more than zero.

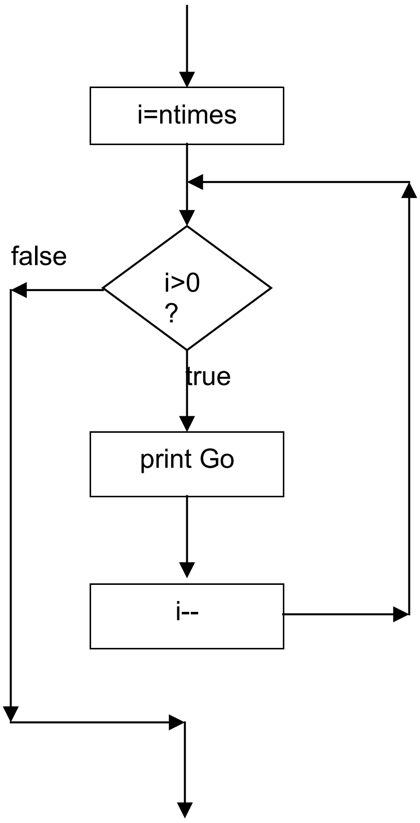


Fig 3 Loop structure

Exercise 1

1 By referring to program *goloop.c* shown above, identify
- **a)** the function call made to *printGo()*
- **b)** the argument value passed to *ntimes*

Recursion

In a recursive function call the function says to **itself**: *hey, function, go and do your job*.

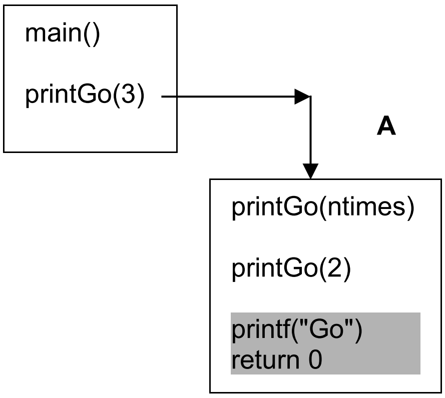
```
/* gorecurse.c - uses recursion to print Go Go Go */

#include <stdio.h>

int printGo(int ntimes)
{
  if (ntimes <= 0) {
    printf("\n");
    return 0;
  }
  printGo(ntimes - 1);   ← recursive function call
  printf("Go ");
  return 0;
}
```

```
int main()
{
  printGo(3);
  return 0;
}
```

Program run

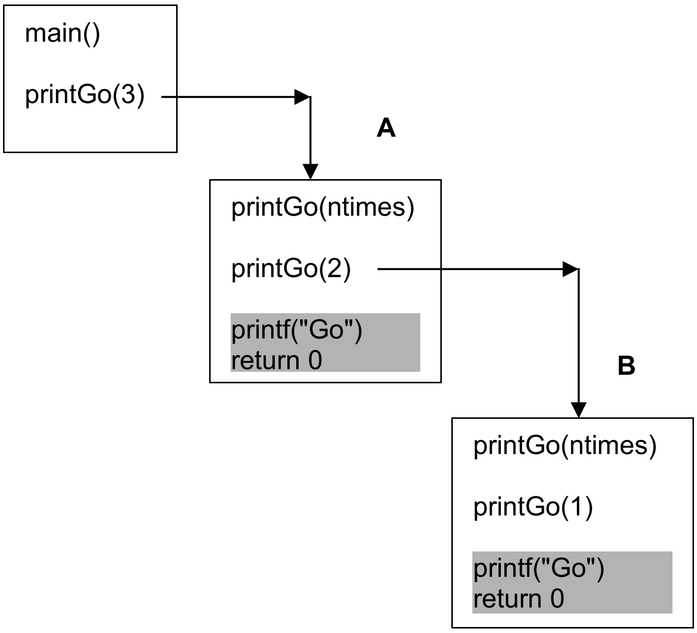
```
Go Go Go
```

We trace the sequence of function calls.

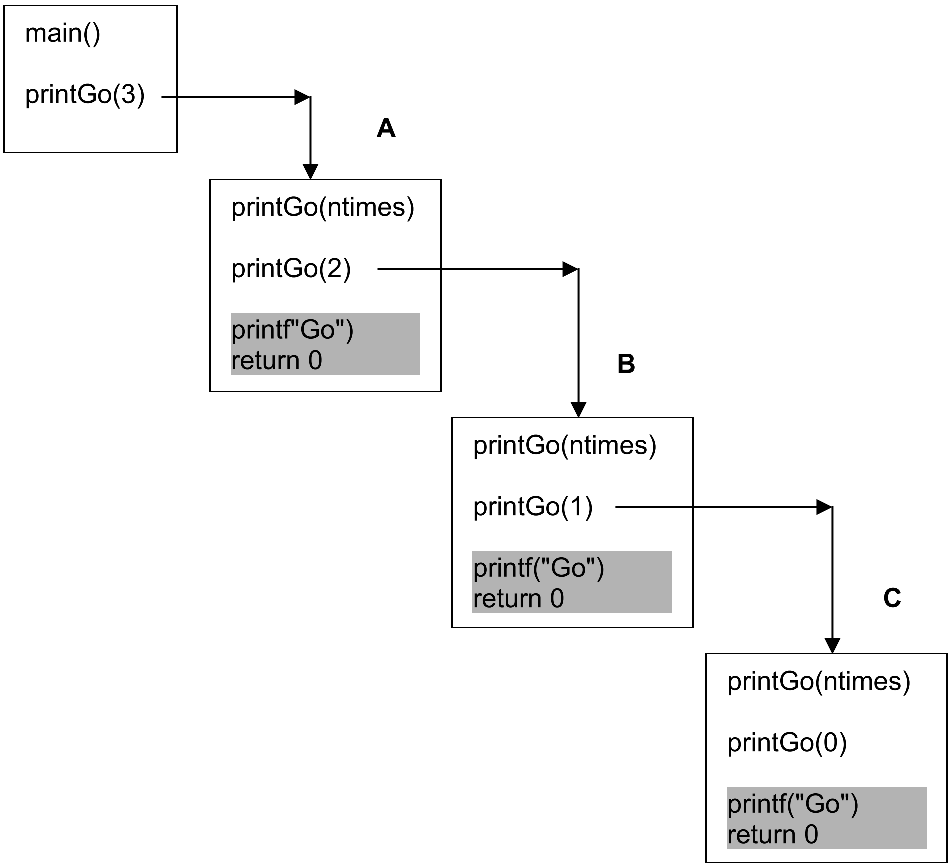


main() makes the first call to *printGo()* and passes the argument value *3*.

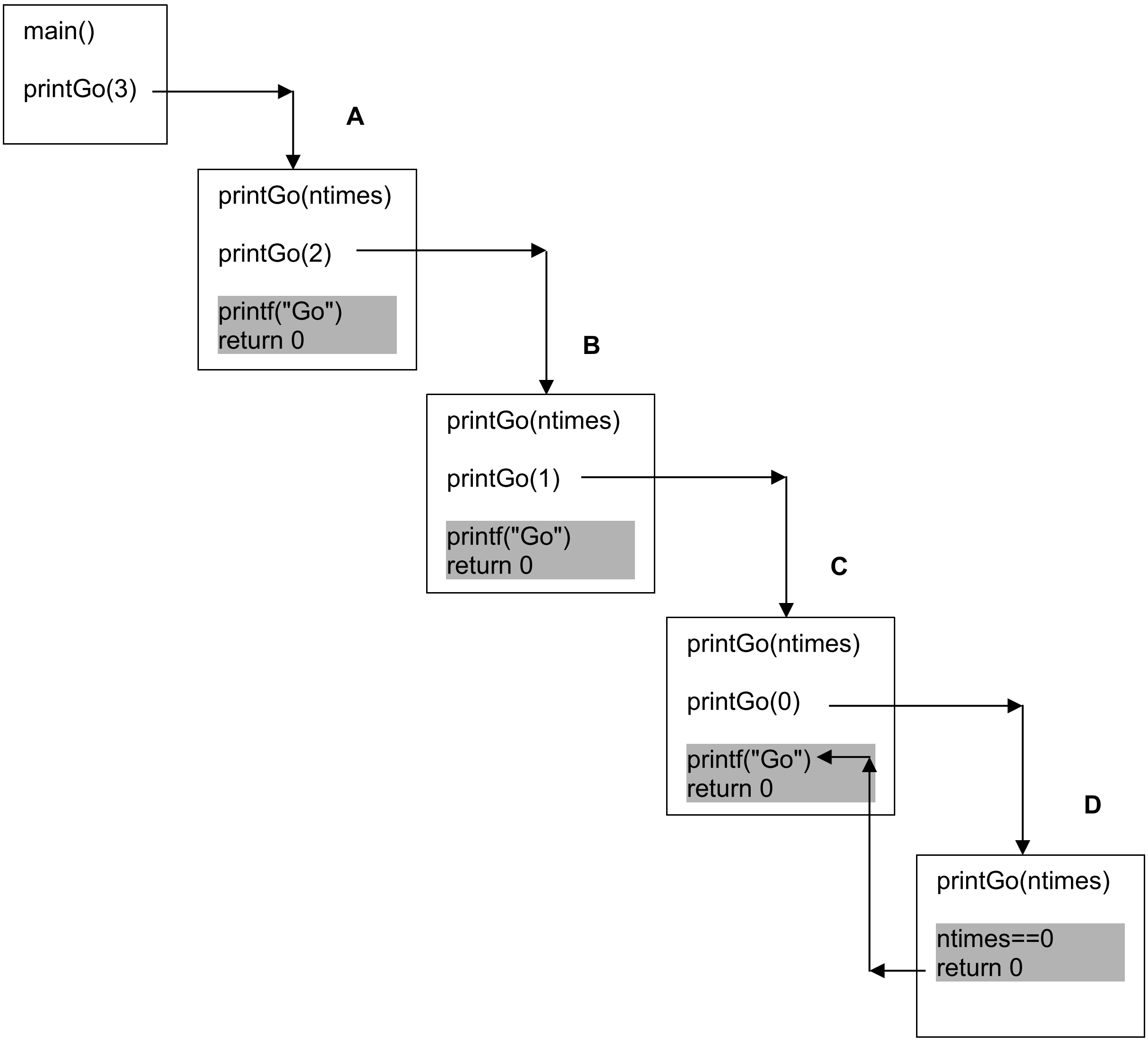
In the *printGo()* function (**A**), the next call to *printGo()* is made. This time 2 is passed to *printGo()*. *printf("Go")* and *return 0* are not reached.



The next version of the *printGo()* function (**B**) receives the argument value *2*, and then calls *printGo()* and passes to it the argument value *1*. *printf("Go")* and *return 0* are not reached.



In the next copy of the *printGo()* function (**C**), a call is made to *printGo()* with the argument value *0*. Again, *printf("Go")* and *return 0* are not reached.



In the next version of the *printGo()* function (**D**), *ntimes* is zero and the function returns zero to its caller. *ntimes == 0* is the stopping condition because recursion stops at this point. Its caller (**C**) prints Go and returns control to its caller.

The sequence of function calls continues to be unravelled, as shown below.

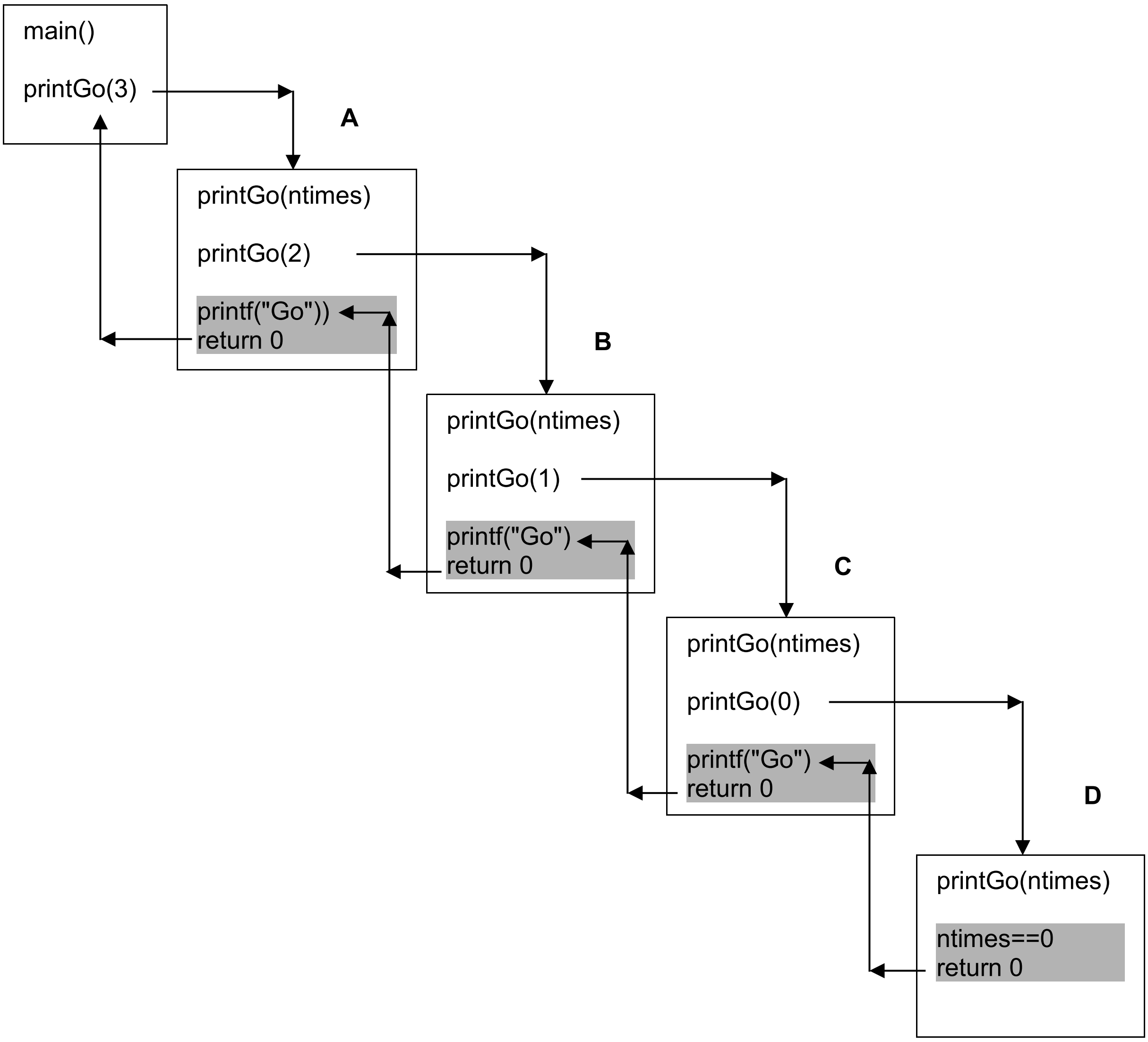


Fig 4 shows the sequence of recursive function calls. Each function call creates a new copy of printGo(). *main()* calls *printGo(3) (**A**). printGo(int ntimes) calls printGo(2) (**B**). printGo(int ntimes) calls printGo(1) (**C**). printGo(ntimes) calls printGo(0) (**D**). In this last version of printGo(), (**D**) ntimes is zero and the function returns control to its caller. Then (**C**) prints Go and returns to its caller. Then (**B**) prints Go and returns to its caller. Then (**A**) prints Go and returns to its caller, main().*

Write Your Own Recursive Functions

Problem 1: using recursion, sum the first five positive integers: 1, 2, 3, 4 and 5. The answer should be 15 (1 + 2 + 3 + 4 + 5 = 15).

1 Understand the Problem. Write out the simplest concrete examples.

```
1                 = 1
1 + 2             = 3
1 + 2 + 3         = 6
1 + 2 + 3 + 4     = 10
1 + 2 + 3 + 4 + 5 = 15
```

2 Write the general case. We notice that the sum of *n* integers is the sum of *n - 1* integers plus *n*.

i.e. *sum(n) = sum(n - 1) + n*

e.g. *sum(5) = sum(4) + 5*

3 Include the stopping and continuation conditions.

if n == 0, sum(n) = 0	*stopping condition*
if n > 0, sum(n) = sum(n - 1) + n	*continuation condition*

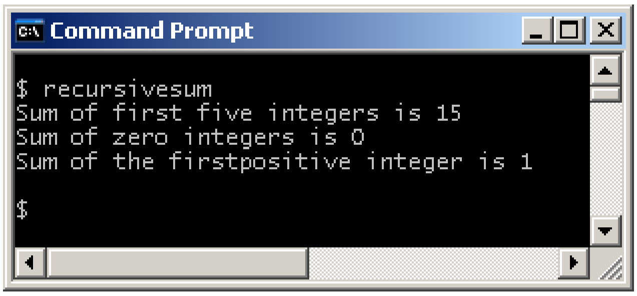
4 Write the recursive function in C

```
int sum(int n)
{
  if (n <= 0)
    return 0;
  return sum(n - 1) + n;
}
```

5 Test the function. Remember to check that the function gives sensible answers at the boundary points e.g. when there is zero, or just one number, to be summed.

```
/* recursivesum.c - uses recursion to find the sum of the
first n positive integers */

#include <stdio.h>

int sum(int n)
{
  if (n <= 0)
    return 0;
  return sum(n - 1) + n;
}


int main()
{
  printf("Sum of first five integers is %d\n", sum(5));
  printf("Sum of zero integers is %d\n", sum(0));
  printf("Sum of the firstpositive integer is %d\n", sum(1));
  return 0;
}
```

Program run:



```
$ recursivesum
Sum of first five integers is 15
Sum of zero integers is 0
Sum of the firstpositive integer is 1

$
```

Problem 2 Using recursion, sum the first five even integers: 2, 4, 6, 8, 10. The answer should be 30.

1 Understand the problem.

n	1	2	3	4	5
term	2	4	6	8	10
sum	2	6	12	20	30

sum: 0 + 2 = 2, 2 + 4 = 6, 6 + 6 = 12, 12 + 8 = 20, 20 + 10 = 30

2 Write the general case. If there *n* integers to sum,

sum(n) = sum(n - 1) + n x 2

e.g. sum(5) = sum(4) + 5 x 2

3 Write the stopping and continuation conditions

if n == 0, sum(n) = 0
if n > 0, sum(n) = sum(n - 1) + 2n

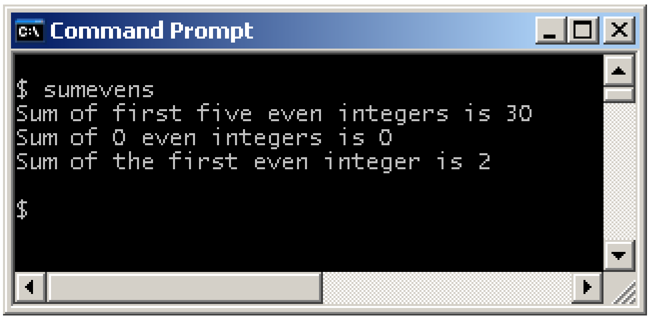
4 Write the recursive function in C.

```
int sumevens(int n)
{
  if (n <= 0)
    return 0;
  return sumevens(n - 1) + n * 2;
}
```

5 Test the function

```
/* sumevens.c - uses recursion to sum the first n even integers */

#include <stdio.h>

int sumevens(int n)
{
  if (n <= 0)
    return 0;
  return sumevens(n - 1) + n * 2;
}

int main()
{
  printf("Sum of first five even integers is %d\n", sumevens(5));
  printf("Sum of 0 even integers is %d\n", sumevens(0));
  printf("Sum of the first even integer is %d\n", sumevens(1));
  return 0;
}
```

Program run



```
$ sumevens
Sum of first five even integers is 30
Sum of 0 even integers is 0
Sum of the first even integer is 2

$
```

Exercise 2

1. Design, write and test a C function that uses recursion to sum the first n odd integers.
2. Design, write and test a C function that uses recursion to print a horizontal line of 10 asterisks.
3. Design, write and test a C function that uses recursion to print out the 13 times table up to 13 x 12.

Writing a recursive function involves defining the stopping and recursive cases, and deciding where processing is to go.

```
if (stopping case)
  return something
else if recursive case {
  some processing before
  recursive call
  some processing after
}
```

Most recursive functions do their processing after the recursive call is made.

Recursion in place of Loops

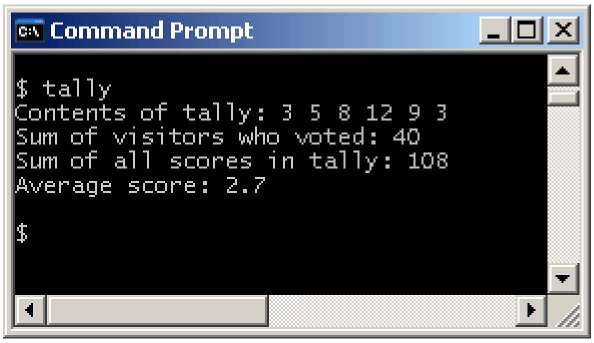
You can usually use recursion instead of a loop.

Problem 3 Visitors rate a website on a scale from 0 (appalling) to 5 (brilliant). A tally of the scores awarded by visitors is held in an array named *tally*. Use recursion to find the average score.

```
/* tally.c - uses recursion to find the average rating of a website
*/

#include <stdio.h>

/* populate: fills array with sample values */
int populate(int tally[])
{
  tally[0] = 3;  /* 3 visitors scored the website 0 points */
  tally[1] = 5;  /* 5 visitors scored the website 1 point */
  tally[2] = 8;  /* 8 visitors scored the website 2 points */
  tally[3] = 12;
  tally[4] = 9;
  tally[5] = 3;
  return 0;
}


/* print: displays the contents of the array */
int print(int tally[], int i)
{
  if (i < 0)
    return 0;
  print(tally, i - 1);
  printf("%d ", tally[i]);
  return 0;
}


/* sumContents: returns sum of array contents */
int sumContents(int tally[], int i)
{
  if (i < 0)
    return 0;
  else
    return sumContents(tally, i - 1) + tally[i];
}


/* sumScores: returns the total scores held in array */
int sumScores(int tally[], int i)
{
  if (i < 0)
    return 0;
  else
    return sumScores(tally, i - 1) + tally[i] * i;
}
```

```
int main()
{
  int tally[6];  /* indexed 0..5 */
  const int lastIndex = 5;

  populate(tally);
  printf("Contents of tally: ");
  print(tally, lastIndex);
  printf("\n");

  printf("Sum of visitors who voted: %d\n",
                                sumContents(tally, lastIndex));
  printf("Sum of all scores in tally: %d\n",
                                sumScores(tally, lastIndex));
  printf("Average score: %0.1f\n", (double)sumScores(tally,
                      lastIndex) / sumContents(tally, lastIndex));

  return 0;
}
```

Program run:



```
$ tally
Contents of tally: 3 5 8 12 9 3
Sum of visitors who voted: 40
Sum of all scores in tally: 108
Average score: 2.7

$
```

Exercise 3

1. Trace the sequence of recursive function calls for
   a) *print()*
   b) *sumContents()*
   c) *sumScores()*
   shown in program *tally.c* above. (You could use paper and pencil.)
2. The temperature inside a glasshouse is recorded every hour for 24 hours. Design, write and test a program that provides statistics: maximum, minimum, average, on a collection of a day's temperature readings. Use recursion instead of loops wherever possible.

Answers

Exercise 1

1a `printGo(3);`
1b 3

Exercise 2

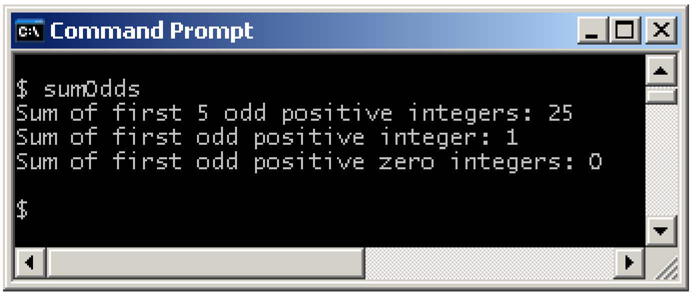
1 To sum the first n odd integers

n	1	2	3	4	5	...	
term	1	3	5	7	9	...	
sum	1	4	9	16	25	...	sum(n) = sum(n - 1) + 2(n - 1) + 1

e.g. sum(5) = 16 + 2 x 4 + 1 = 25
sum(4) = 9 + 2 x 3 + 1 = 16
sum(3) = 4 + 2 x 2 + 1 = 9

Stopping condition: if n == 0 sum(n) = 0
Continuation condition: if n > 0, sum(n) = sum(n - 1) + 2(n - 1) + 1

```
/* sumodds.c - uses recursion to sum the first n positive odd
integers */

#include <stdio.h>

int sumOdds(int n)
{
  if (n <= 0)
    return 0;
  return sumOdds(n - 1) + 2 * (n - 1) + 1;
}

int main()
{
  printf("Sum of first 5 odd positive integers: %d\n",
                                                 sumOdds(5));
  printf("Sum of first odd positive integer: %d\n",
                                                 sumOdds(1));
  printf("Sum of first odd positive zero integers: %d\n",
                                                 sumOdds(0));
  return 0;
}
```



```
$ sumOdds
Sum of first 5 odd positive integers: 25
Sum of first odd positive integer: 1
Sum of first odd positive zero integers: 0

$
```

2

```
/* printStars.c - prints a line of asterisks */

#include <stdio.h>

int printStars(int n)
{
  if (n <= 0)
    return 0;
  printStars(n - 1);
  printf("*");
  return 0;
}

int main()
{
  printStars(10);
  return 0;
}
```

3

```
/* 13xtable.c - uses recursion to print 13 times table */

#include <stdio.h>

int printTable(int m, int n)
{
  if (n <= 0)
    return 0;
  printTable(m, n - 1);
  printf("%d x %d = %d\n", m, n, (m * n));
  return 0;
}


int main()
{
  printTable(13, 12);
  return 0;
}
```