14 Routines and functions

A C program has the following general structure

```
#include etcetera


main ( with or without command-line arguments )
{
  declare variables used (int, char, etcetera)

  perform calculations
}
```

The calculations involve arithmetic computations, etcetera, and certain *functions or routines* such as `atoi(), scanf(), printf()`, which make the work a lot easier. It would be almost impossible to write long programs without being able to write our own functions and routines.

A C program would then look like

```
#include etcetera

<function or routine A> ( <arguments> )
{
... etcetera ...
}

<function or routine B> ( arguments )
{
... etcetera ...
}

...etcetera...

main ( with or without command-line arguments )
{
  declare all variables used (int, char, etcetera)

  perform calculations
}
```

Now the calculations in `main()` can use the functions and routines. For example, we can write a function which calculates the `gcd` of two numbers. It has two *arguments*, resembling the arguments to `main()`.

```
int gcd ( int n, int m )
{
  int x,y,z;
  x = n;
  y = m;
  while ( y > 0 )
  {
    z = x % y;
    x = y;
    y = z;
  }

  return x;
}
```

This is a *function* with two integer *arguments* which returns an integer *value*.

```
#include <stdio.h>

int gcd ( etcetera )
{ as above }

main ()
{
  int n,m,g;
  while ( scanf ( "%d %d", &n, &m ) == 2 )
  {
    g = gcd ( n, m );
    printf ( "gcd (%d, %d) is %d\n", n, m, g );
  }
}
```

Sample session:

```
% gcc g.c
% a.out
1 2
gcd (1, 2) is 1
1001 1261
gcd (1001, 1261) is 13
1261 1001
gcd (1261, 1001) is 13
64 192
gcd (64, 192) is 64
CTRL-D
%
```

Example of a routine to clear specific parts of a 2-dimensional array

```
void clear_array ( int m, int n, double a[10][10] )
{
  int i,j;
  for (i=0; i<m; ++i)
  for (j=0; j<n; ++j)
  { a[i][j] = 0; }
}
```

Useful labour-saving device: redirected input. This has been mentioned already. You can prepare the input in a separate file, call it `temp`, say:

```
1 2
1001 1261
1261 1001
64 192
```

You don't worry about CTRL-D to end the input: the system will recognise the end-of-data in some other way. Then just type

```
a.out < temp
or, to save the output  in a file ‘newtemp’
a.out < temp > newtemp
```

- This gcd function seems to be written the same way as `main()`.

  It is, except for that `int` at the beginning, and it includes a `return` statement which returns the value of `x`.

- Why doesn't `main()` have `int` or something in front of it?

  It should. In the old days it didn't: I'm breaking some convention. Leaving it out doesn't seem to do any harm.

- `Scanf()` returns a value, the number of items read. Does that mean `scanf()` is a function?

  Yes.

- What about `printf()`? Does it return a value?

  No, printf is a *routine,* not a function.

Here is another example of a function. It works for any year in the Gregorian Calendar (1582 onwards), and also for this century, given `yy` is between 0 and 99. The correction in the Gregorian Calendar over the Julian was to make only one century in four a leap year.

```
int is_leap_year ( int yy )
{
  if ( yy % 4 != 0 )
    return 0;
  else if ( yy % 100 != 0 )
    return 1;
  else if ( yy % 400 != 0 )
    return 0;
  else
    return 1;
}
```

So we come to routines. The only difference between routines and functions is that a routine begins with the keyword **void**. This indicates that nothing is returned. For example, `speak()` is a routine:

```
#include <stdio.h>

void speak ( int hello )
{
  if (hello != 0)
    printf ("hello\n");
  else
    printf ("goodbye\n");
}

main()
{
  speak ( 1 );
  speak ( 0 );
}
```

14.1 Simulating routines and functions.

If it is short, a routine or function can be traced out by tabulating the values of its variables (including the arguments).

For example,

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

int xxx ( int x )
{
  int y = 1;
  while ( x/10 > 0 )
  {
    y *= 10;
    x /= 10;
  }

  return y;
}

void yyy ( int x )
{
  int y = xxx ( x );
  int a;

  printf("x=%d, y=%d\n", x, y);

  while ( y > 0 )
  {
    a = (x/y) % 10;
    printf("%d",a);
    y = y/10;
  }
  printf("\n");
}

main(int argc, char * argv[])
{
  int x = atoi ( argv[1] );
  yyy ( x );
}
```

Suppose the input is 12345.
First, xxx

```
            x        y        x/10
        12345
                     1        1234
                    10
         1234                  123
                   100
          123                   12
                  1000
           12                    1
                 10000
            1                    0
returns 10000
```

```
Now, yyy
            x        y        x/y      a
        12345
                 10000
                                1      1
print 1
                  1000
                               12      2
print 2
                   100
                              123      3
print 3
                    10
                             1234      4
print 4
                     1
                            12345      5
print 5
                     0
print "\n"
```

The next example illustrates **recursion,** where a routine calls itself. *How* it works will be explained later.

```
int factorial ( int n )
{
  if ( n == 0 )
    return 1;
  else
    return n * factorial ( n-1 );
}
```

Here is how `factorial(3)` is calculated

```
main()
{
  int n = factorial(3); printf("%d\n", n);
}
The statement
  n = factorial(3) is begun; factorial(3) needs to be
    evaluated.

  factorial(3):
    3 != 0.
    evaluate 3 * factorial(2).
```

```
      factorial(2):
        2 != 0.
        evaluate 2 * factorial(1).
          factorial(1):
            1 != 0.
            evaluate 1 * factorial(0).
              factorial(0):
                0 == 0.
                return 1 as factorial(0)
              factorial(0) completed.
            1 * factorial(0) = 1.
            return 1 as factorial(1).
          factorial(1) completed.
        2*factorial(1) = 2.
        return 2 as factorial(2).
      factorial(2) completed
    3*factorial(2) = 6
    return 6 as factorial(3)
  factorial(3) completed
  n becomes 6 and 6 is printed.
```

Here is another recursive example.

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

void yyy ( int x )
{
  if ( x > 9 )
    yyy ( x/10 );

  printf("%d", x%10);
}

main(int argc, char * argv[])
{
  int x = atoi(argv[1]);
  yyy ( x );
  printf("\n");
}
```

Three more questions.

- Can one *write* a function inside another? The answer is ‘yes,’ but it is unnecessary.
- Can one *use* a function or routine A in some other one B, not just `main()`? Answer: yes, so long as A appears before B in the program.

- What if A is written after B? One can include a *function prototype* for A, before B.

A function prototype is just a function definition with the body (the part between curly braces) replaced by a semicolon.