12 Functions and routines; simulation

12.1 Preview

Routines and functions will be introduced. The important ideas connected with them are

- Return type — routines have 'return type' `void`.
- Function/routine arguments
- Local variables
- Calling a function/routine

12.2 Routines

Up to now, all programs have fitted the pattern

```
#include ...
int main ( ... )
{
...
}
```

and all the work is crammed into the 'main' part. This would be very difficult if the program performs a complex task. In that case, the work should be divided into smaller pieces and each piece should be performed within a single *function or routine.*

We have used several functions and routines already.

```
atoi () a function
atof () a function
scanf() a function
printf() a routine
```

The difference between function and routine is that a function *returns a value* whereas a routine performs a task without returning a particular value.

The `main` procedure[1] is actually a function:

```
int main ( int argc, char *argv[] )
{
}
```

If you remember to put `return` statements in the `main` procedure, the value returned can be picked up by the operating system. Usually the value returned indicates successful/unsuccessful termination of the program.

The function `main()` has two arguments, `argc` and `argv`.

Every function or routine follows the same pattern as `main`:

[1] Procedure is another word for routine.

```
<return type> <name> ( <argument list> ), e.g.,
<return type> = ‘int’ <name> = ‘main’ ( <arg list> = ‘int argc, char *argv[]’ )
{
}
```

where the return type can be `int`, `char`, `char*`, etcetera.

A routine is the same *except that* the return type is `void`:

```
void <name> ( <argument list> )
{
}
```

12.3 Little example of a routine

```
#include <stdio.h>
void show ( int n )
{
  printf("n is %d\n", n);
}
int main()
{
  show(45);
}
```

12.4 GCD example.

The following code implements Euclid's gcd algorithm. Note the 'two at a time' comment, where `scanf` is reading two numbers at a time. The code reads pairs of numbers and prints their gcd. Remember that the value returned by `scanf()` is the number of items read, if at least one item has been read.

More comments could be added later to explain how the function works.

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

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

```
int main ( )
{
  int m,n;
  while ( scanf("%d %d", &m, &n) == 2 ) // two at a time
  {
    printf("gcd(%d,%d) is %d\n", m,n,gcd(m,n));
  }
}
%cat forgcd
1 2 -3 4 6 8 15 35 60 -90
%gcc simplegcd.c
%a.out < forgcd
gcd(1,2) is 1
gcd(-3,4) is 1
gcd(6,8) is 2
gcd(15,35) is 5
gcd(60,-90) is 30
%
```

12.5 Arguments and local variables

To avoid repeating 'routine or function,' 'routine' will mean both.

Routines in general have arguments, like `m,n` in the `gcd` routine, and *local variables* like `x,y,z` in the same routine.

The names[2] `x,y,z` may occur in many other routines (in the same program), but they have *no connection* with any other variable of the same name. That's why they're called local variables.

The arguments like `m,n` can be treated like local variables. Their names do not connect them with any other variable or argument in other routines.

12.6 Calls to routines and functions

```
From above:
    printf("gcd(%d,%d) is %d\n", m,n,gcd(m,n));
```

The expression

```
gcd(m,n)
```

in the above line is a ***call*** to the function `gcd(m,n)`. The following happens.

- The values of `m,n` in the `main()` procedure are ***copied*** to the arguments `m,n` of the `gcd()` procedure. The variables `m,n` in `main()` have the same names as in `gcd()`, but that is a coincidence.

[2]Technically called 'identifiers.'

- The statements in `gcd()` are executed as given, and the final value of `x` is returned.
- The value returned is what is printed as `gcd(m,n)`.

12.7 Another version of `gcd()`

It is said that the arguments `m,n` behave as local variables. The local variables `x,y` are not required.

One could revise the function as follows:

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

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

The resulting program gives the same output as the original. But look at the call `g = gcd(m,n);` When the function is called, the *argument* `m` is changed and actually is returned as the `gcd()` value, assigned to `g`. But the `m` in the main program is *not* changed, as the printout proves, or will if you compile and run the program. This is because the two variables are completely different; they have the same name, but that is a coincidence.

The versions where `m,n` are used as local variables save a tiny amount of space, but the code is harder to understand, or to discuss.

12.8 Boolean functions

In C (at least, in its original form), there is no special boolean type (taking the values true/false). Expressions formed with the logical connectives and relations are actually *evaulated*, to integers. Conventionally, 1 means true and 0 means false.[3]

[3] Actually 0 means false and any nonzero integer is interpreted as true.

So here is a piece of code which assumes $0 \leq$ `yy` ≤ 99 and `yy` is a year in this century, and returns 1 if a leap year and 0 otherwise.

```
int leapyear ( int yy )
{
  return yy % 4 == 0;
}
```

Exercise: rewrite `leapyear()` to allow for any any `yy` ≥ 1582.

Here is a similar example.

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

int divides ( int m, int n )
{
  return (m != 0 ) && ( n % m  == 0 );
}

int main ( int argc, char * argv [] )
{
  int m = atoi ( argv[1] ), n = atoi ( argv[2] );
  if ( divides ( m, n ) )
    printf("%d divides %d\n", m, n);
  else
    printf("%d does not divide %d\n", m, n);
}
```

12.9 An array argument

We have seen this already: `argv[]`. The next example shows numbers being read into an array, and the array being passed to a function `total()`.

C, at least in its original form, pays no attention to the size of an array, the amount of memory reserved for the array. So it is necessary to communicate the number of array entries to be added: `int total ( int n, int a[] )`. The argument `n` is the number of array entries to be totalled.

Notice that the size of the array `a[]` is not given. The function will add `n` array entries, whether or not that equals the array size. Compare with `int main ( int argc, char * argv[] )`.

```
% cat addup-array.c
#include <stdio.h>
int total ( int n, int a[] )
{ int s = 0;
  int i;
  for (i=0; i<n; ++i)
  { s = s + a[i];
  }
  return s;
}

int main ( )
{ int array[1000];
  int count, x;
  count =0;
  while ( count < 1000 && scanf( "%d", &x ) == 1 )
  { array[count] = x;
    ++ count;
  }
  printf("%d numbers total %d\n", count, total(count, array));
}
% gcc addup-array.c
% a.out
3 1 4 1 5 9 2 6 5
9 numbers total 36
%
```

12.10 Simulating the `gcd()` function

Simulation often helps one to understand how a routine works. For example, let us simulate `gcd(63,35)`, with the original code.

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

You should show the values taken by `m,n,x,y,z`. It is not necessary to recite each statement, so long as the effect of the statement is clear. But also we should include the value of the

condition `y > 0`, since it controls the while-loop. The condition evaluates to 1 or 0, but it is clearer to write `yes` or `no`.

Also, the presentation should be *staggered* so that the order in which the statements are executed is clear. (The order of the three assignment statements in the while-loop is crucial.)

```
      m    n   x   y   z  y>0
     63   35
              63
                  35
                          yes
                      28
              35
                  28
                          yes
                       7
              28
                   7
                          yes
                       0
               7
                   0
                          no
    return 7
```

12.11 Simulating the `total()` function

```
int total ( int n, int a[] )
{ int s = 0;
  int i;
  for (i=0; i<n; ++i)
  { s = s + a[i];
  }
  return s;
}
```

For example, totalling an array {3,1,4} with `n==3`. The arguments are `3,  {3,1,4}`, and the local variables are `s,i` We also show `a[i]` and the condition `i<n`.

```
    n     a               s      i      i<n   a[i]
    3     {3,1,4}
                          0
                                 0
                                        yes
                                               3
                          3
                                 1
                                        yes
                                               1
                          4
                                 2
                                        yes
                                               4
                          8
                                 3
                                        no
return 8
```