Functions + routines.

$$\underset{\uparrow \atop \langle \text{return type} \rangle}{\text{int}} \quad \underset{\uparrow \atop \langle \text{name} \rangle}{\text{main}} \quad (\underbrace{\text{int argc, char *argv[ ]}}_{\langle \text{arguments} \rangle})$$

{ body of fn/ routine :
<local variables>, <statements> }

a routine is a function whose return type is void (does something, returns nothing).

atoi ()
atof()
scanf () } functions

printf : routine.

Trivial example

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

GCD:

```
#include<stdio.h>
#include<stdlib.h>
int gcd(int m, int n)
{ int x = abs(m), y=abs(n), 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)
  { printf("gcd(%d, %d) is %d\n",
            m, n, gcd(m,n));
  }
}
```

Arguments and local variables.

in gcd function   m, n are arguments

x, y, z are local variables

Variables and arguments

in other routines (short for routine/function)

sharing any of these names m, n, x, y, z,

are completely independent of them.

CALLING A ROUTINE

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

call to gcd

(1) m, n in main() copied to m, n in gcd(). COINCIDENCE.

(2) gcd is executed, final value of x returned.

(3) that value is printed.

---

§12.7 makes clear that m in main() and in gcd() are independent.

§12.9 Boolean functions.

In (original) C, no boolean true/false.
Int values used

$$\begin{cases} 0 & = \text{false} \\ \text{nonzero} & = \text{true} \\ & \quad\text{(usually, 1)} \end{cases}$$

EG

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

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

§12.9 Routine with array arguments

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

  return s;
}           // returns a[0]+...+a[n-1]
```

Simulate gcd(63, 35)

while (y > 0) { z = x % y; x = y; y = z; } return x; ...

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

return 7

Simulate total(3, {3,1,4})

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

n	a	s	i	i<n?	a[i]
3	{3,1,4}				
		0			
			0	Y	3
		3			
			1	Y	1
		4			
			2	Y	4
		8			
			3	N	

return 8

Simulate gcd(63, 35)

while (y > 0) { z = x % y; x = y; y = z; } return x; ...

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

Simulate total(3, {3,1,4})

... for (i=0; i<n; ++i){ s+=a[i];} return s

n	a	s	i	i<n?	a[i]
3	{3,1,4}				
		0			
			0	Y	3
		3			
			1	Y	1
		4			
			2	Y	4
		8			
			3	N	

return 8

Routines which call themselves: RECURSIVE

```
int gcd(int m, int n)
{ if (n == 0)
  {return m;}
  else
  {return gcd(n, m%n);}
```

```
int total (int n, int a[])
{ if (n <= 0)
    {return 0;}
    else
    {return total(n-1, a) + a[n-1];}
}
```

VARIOUS RECURSIVE ROUTINES

```
int gcd( int m, int n)
{if (n == 0)
  { return m;}
  else
  { return gcd(n, m % n);}
}
```

$$gcd(m,n) = gcd(n, m \% n)$$

```
int total(int n, int a[])
{ if (n == 0)
   {return 0;}
   else
   {return total( n-1, a)
              + a[n-1];
   }
}
```

$$\sum_{i<n} a_i = \left(\sum_{i<n-1} a_i\right) + a_{n-1}$$

```
int rec_print(int n)
{ if (n>0)
  { rec_print (n/10);
    printf("%d", n%10);
  }
}
```

$n = (a_k \cdots a_0)_{10}$

$n/10 = (a_k \cdots a_1)_{10}$

$n\%10 = a_0$

'RUSSIAN PEASANT ALGORITHM'

```
int product (int m, int n)
{ if (n==0)
   { return 0;}
   else
   {  int p= product
              (m, n/2);
      if (n%2 == 0)
      { return p+p;}
      else
      {return p+p+m;}
   }
}
```

If n is even, $n=2k$ say

then $mn = mk + mk$

If odd, $2k+1$ say

then

$mn = mk + mk + m$.

similar

method

for A^k

PRIME FACTORS.

```
void factorise( int n)
{ int seems_prime = 1 ;
  int m;
  for (m= 2; seems_prime
          && m<n; ++m)
  { if(divides( m, n))         <- smallest divisor
                                  m is prime
    { seems_prime = 0;         <- n is composite
      printf(" %d", m);
      factorise( n/m);         <- recurse
    }
  }
  if (seems_prime)
  {printf(" %d", n);}          <- n is prime.
}
```