Mathematics 1261 (computing with C), Michaelmas 2012

Colm Ó Dúnlaing

March 31, 2014

Syllabus

1 Programming languages

The course aims at a reasonable level of skill in C programming.

Many computer programming languages were invented in the 50s and 60s. The original idea was — I think — that a programming language would be so close to English that non-experts could use it.

(1.1) A computer accepts instructions in a very compact form called its *machine code.* A *machine program* (also called a ‘binary’ or ‘executable’) is a list of instructions in machine code. In the 1970s, with small microprocessors, it was common to write programs directly in machine code. Here is an example of machine code. Nowadays, most machine-code programs have thousands of lines like these.

```
Memory    Machine
Address   instructions------------------------------------

00000210  69 6e 5f 75 73 65 64 00  5f 5f 6c 69 62 63 5f 73
00000220  74 61 72 74 5f 6d 61 69  6e 00 47 4c 49 42 43 5f
00000230  32 2e 30 00 00 00 02 00  02 00 01 00 00 00 00 00
00000240  01 00 01 00 24 00 00 00  10 00 00 00 00 00 00 00
00000250  10 69 69 0d 00 00 02 00  56 00 00 00 00 00 00 00
00000260  d8 95 04 08 06 05 00 00  d0 95 04 08 07 01 00 00
00000270  d4 95 04 08 07 02 00 00  55 89 e5 83 ec 08 e8 61
00000280  00 00 00 e8 c8 00 00 00  e8 f3 01 00 00 c9 c3 00
00000290  ff 35 c8 95 04 08 ff 25  cc 95 04 08 00 00 00 00
000002a0  ff 25 d0 95 04 08 68 00  00 00 00 e9 e0 ff ff ff
000002b0  ff 25 d4 95 04 08 68 08  00 00 00 e9 d0 ff ff ff
000002c0  31 ed 5e 89 e1 83 e4 f0  50 54 52 68 20 84 04 08
000002d0  68 c0 83 04 08 51 56 68  84 83 04 08 e8 bf ff ff
```

(1.2) A kind of low-level programming language called *assembler language* was developed to make programming easier. It was much easier than machine code, but still very laborious. Here is an example.

```
        .file       "prog1.c"
        .section        .rodata
.LC0:
        .string        "%d\n"
        .text
.globl  main
        .type        main, @function
```

```
main:
        pushl       %ebp
        movl        %esp, %ebp
        subl        $24, %esp
        andl        $-16, %esp
        movl        $0, %eax
        subl        %eax, %esp
        movl        $5, -4(%ebp)
        leal        -4(%ebp), %eax
        incl        (%eax)
        movl        -4(%ebp), %eax
        movl        %eax, 4(%esp)
        movl        $.LC0, (%esp)
        call        printf
        leave
        ret
        .size        main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident        "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-13)"
```

Assembler languages are very hard to use nowadays, probably because they are almost never used. In the 60s and 70s they were easier to use.[1]

Next came programming languages which made programming much easier. Here are some.

- FORTRAN was suitable for scientific calculation, especially involving matrices. Since FORTRAN is relatively simple, it can do its work efficiently and is still used for numerical computation.
- COBOL is still used for commercial programming, such as payroll management. It was designed in the 50s by one woman, Grace Hopper, from the US Navy.
- APL is a weird but concise language which may still have its enthusiasts. Its major advantage was conciseness, a consideration when using slow modems but now irrelevant. As it uses many special symbols, it requires an APL keyboard.
- LISP was developed for artificial intelligence.
- ALGOL was a more sophisticated substitute for FORTRAN.
- SIMULA67 was the first 'object-oriented' language, leading to SmallTalk, C++, Eiffel, and Java.
- PL/1 was IBM's language intended to combine the power of both Fortran and Cobol.
- ALGOL68 was, at the time, a very sophisticated language. At the time people found it very difficult to implement.

[1] I have been told that Motorola 6800 assembler is much easier than Intel assembler, which is shown here.

- BASIC was probably meant for children, but became important because it was easy to implement on minicomputers.
- PASCAL was a much-simplified form of ALGOL (and ALGOL68). Unlike ALGOL68, it was easy to implement on minicomputers, which made it very popular in the 70s and 80s.
- C was introduced in the early 70s. It was possibly meant to be a form of PASCAL without certain limitations. C programs are unmistakable in appearance, quite unlike Pascal programs, but only at the level of programming notation.
- ADA was developed by the US military in the early 1980s, to replace several different programming languages used in administration, numerical work, and bombs. In appearance it is closer to Pascal than C, though with a notation different from both.
- C++ was developed in the 80s as an object-oriented form of C. It closely resembles C.
- EIFFEL was developed in the 80s at the same time as C++. It resembles Ada. Some respected authorities consider Eiffel better than C++ and Java.
- JAVA was originally meant (I believe) for programming toasters, but is now a very popular C-like variant of Eiffel (and C++), with internet applications.
- SETL, developed in the late 70s, is a little-known language for writing algorithms with a notation based on set theory.
- SNOBOL is an old language intended for processing textual data. It may have had some influence on special-purpose languages like Awk, Perl, and Python.
- FORTH is a language I never saw, but its programs are in postfix form, which has sometimes been used on calculators.
- POSTSCRIPT is a graphics and typesetting language which also uses a postfix notation.
- TeX also provides a complete, but low-level, programming language.

(1.3) Here is a simple C program — actually, the assembler program above was derived from it.

```
#include <stdio.h>
      /* first program example */
main()
{
  int i=5;
  i = i+1;
  printf ("%d\n", i);
}
```

It is trivial, but uses enough C to be able to guess what some of the assembler code does.

- `i` is an integer, initialised to 5.
- Next `i` is replaced by `i+1`, i.e., 6.
- Last, `i` is printed. The `"%d\n"` is a *format control* string. What does `\n` mean?
- What is `#include <stdio.h>` good for?
- What is `/* first program example */` good for?

Here are some examples of a program to scan a list of parentheses and say whether the list is balanced. For example, ()(()) is balanced but ())()(() is not.

Little if any of this code has been tested; none of the obsolete languages and none of the exotic languages. The interested student is invited to test it.

```
FORTRAN
      FUNCTION BALANCED ( INTEGER S [100], INTEGER SIZE )
      INTEGER SURPLUS
      SURPLUS=0
      I=0
LOOP  I=I+1
      IF I.LE.SIZE GOTO NEXT
      IF SURPLUS.EQ.0 RETURN TRUE
      RETURN FALSE
NEXT  IF S[I].EQ.RPAREN GOTO RPAR
      SURPLUS=SURPLUS+1
      GOTO LOOP
RPAR  SURPLUS=SURPLUS-1
      IF SURPLUS.GE.0 GOTO LOOP
      RETURN FALSE
RPAREN EQ 1H')'
```

```
PASCAL
function balanced ( s: packed array [1..100] of char;
                    size: integer ): boolean;
   var
     i, surplus: integer;
     result: boolean;
begin
  surplus := 0;
  balanced := true;

  for i := 1 to size do
  begin
   if s[i] = ')'
     surplus := surplus - 1;
   else
     surplus := surplus + 1;
   if surplus < 0 then
     balanced := false;
  end

  if surplus <> 0 then
    balanced := false
end
```

```
C
int balanced ( char s[] )
{
  int i, surplus;

  surplus = 0;

  for (i=0; s[i] != '\0'; ++i)
    if (s[i]=='(')
      ++surplus;
    else
    {
      -- surplus;
      if (surplus<0)
        return 0;
    }

  return ( surplus == 0 );
}
```

```
EIFFEL
balanced (s: STRING) : BOOLEAN is
  local
    i, surplus: INTEGER
  do
    from
      i := 1
    until
      i > s.count or surplus < 0
    loop
      if s.item[i] = '(' then
        surplus := surplus+1
      else
        surplus := surplus-1
      end
      i := i+1
    end

    Result := (surplus = 0)
  end
```

```
LISP
define ((
 (right_to_left ( lambda s )
    (cond ((null s) 0 )
          ((negp (right_to_left (cdr s))) -1)
          ((eq (car s) '(') (sub1 right_to_left (cdr s)))
          (T (add1 right_to_left (cdr s)))
    )
 )
 (balanced (lambda s) (zerop (right_to_left s)))
))
```

APL uses a special keyboard, so we use mathematical typesetting:

$$A \leftarrow S =' (' - S =')'$$
$$B \leftarrow +\backslash A$$
$$C \leftarrow (B \geq 0) \wedge B[\rho B] = 0$$

2 Computers and number systems

All computer data is stored as patterns of 0s and 1s. A 'bit' is a binary digit, i.e., 0 or 1, or an object which can take these values. There is a multiplicative effect, so that 8 bits combined together can take $2^8 = 256$ different values.

A computer has several components, including **Central memory, central processor, hard disc, and terminal (or monitor).**

Long-term data is on the hard disc; the central processor works on short-term data in the central memory.

Here is a C program

```
#include <stdio.h>

main()
{
  printf("Hello\n");
  printf("there\n");
}
```

Create a file `hello.c` containing the above lines, then run

```
gcc hello.c
```

This will create a file `a.out` which the computer can run as a program:

```
a.out
```

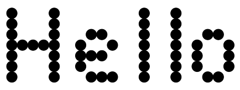
will cause the message

```
Hello
there
```

to be written to the terminal.

Question: what's the '`\n`' for?

(2.1) Although letters on the terminal look like ordinary newsprint, say, under close inspection the letters spelling `Hello` are just patterns of dots, something like



How are these letters stored on a computer? they could be stored as 7×5 patterns of zeroes and 1s, where 0 means 'no dot' and 1 means 'dot.' This would require 35 bits per letter. Instead, all characters are stored as 8-bit patterns under an internationally agreed code, the ASCII code. To learn more, type

```
man ascii
```

ASCII code for H is 01001000, for e is 01100101, and so on. Figure 1 shows the basic components.

Conclusions. The computer stores all data as patterns of 0s and 1s, called **bitstrings.** All characters appear on the screen as patterns of dots.

Central memory, processor, hard disc. When you have edited and saved your program `hello.c`, it is now stored on the **hard disc.** (in ASCII, of course). It is *data.*

It is the **processor** which does the work of the computer. Its job is to read **instructions** from central memory and execute them. The instructions are contained in **executable programs.**

When you type

```
gcc hello.c
```

the computer copies an executable program called **gcc** into central memory, then executes that program on the data contained in `hello.c`. It produces a new executable program which is usually called **a.out** and stores it on disc.

When you type

```
a.out
```

the computer copies `a.out` into central memory and executes it, with the results as described.

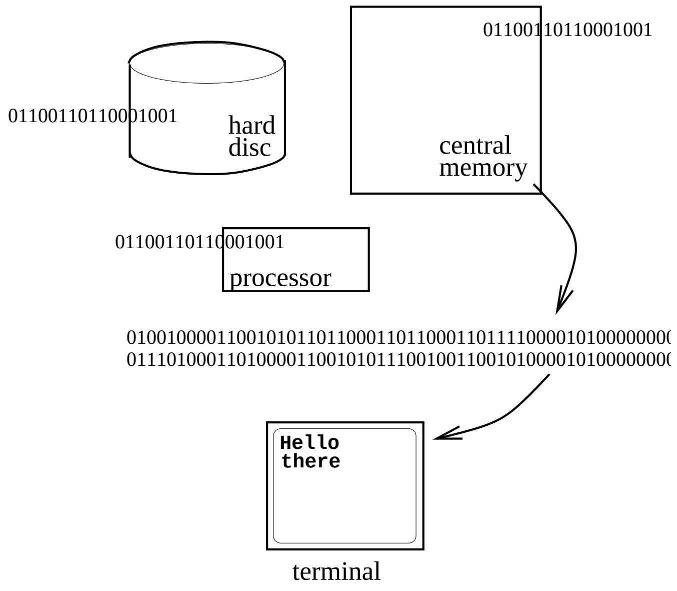


Figure 1: Parts of a computer

2.1 Number systems

Our decimal number system is derived from the human hand.[2] Binary numbers are much simpler.

The binary string `01001000` represents

$$0 + 0 \times 2 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 = 8 + 64 = 72$$

(that is, 72 in decimal, of course).

It is easy to list the binary strings of length 3 in ascending order:

```
000, 001, 010, 011,
100, 101, 110, 111
```

The rightmost bit is called the 'low-order bit.' The 'low order bit' changes most often; the next bit changes half as often; the high-order bit changes only once.

It is easy to convert a bitstring into an *octal string.* Simply put it in groups of 3, starting from the right. Thus

```
01001000
01 001 000
 1   1   0
```

The ASCII code for H has octal value 110. On the other hand, interpreting 110 as an octal string we get

[2]Or from any primate's.

$$0 + 1 \times 8 + 1 \times 8^2 = 72$$

(again, 72 in decimal).

Question. The ASCII code for e is 01100101 as a bitstring. What is it in octal? in decimal?

Octal numbers give a compact way to represent bitstrings. So do **hexadecimal numbers**, which are numbers to base 16. We need 16 **hex digits** to form hexadecimal numbers. One uses `a,b,c,d,e,f` (or `A,B,C,D,E,F`) for the digits ≥ 10. Every hex digit equals four binary digits. The hex digits convert to octal, binary, and decimal as follows

hex	octal	binary	decimal
0	0	0000	0
1	1	0001	1
2	2	0010	2
3	3	0011	3
4	4	0100	4
5	5	0101	5
6	6	0110	6
7	7	0111	7
8	10	1000	8
9	11	1001	9
a	12	1010	10
b	13	1011	11
c	14	1100	12
d	15	1101	13
e	16	1110	14
f	17	1111	15

There are procedures for addition, subtraction, multiplication, and division, in binary, octal, and hex. Addition is easy. For example,

```
    binary    octal   hex    Decimal

  10100011     243     a3      163
 +11010101    +325    +d5     +213
----------    ----    ---     ----
        10      10      8      376
       10       7     17
      10       5      ---
     1        ----    i.e
    1         i.e.     c3
   1           243    +d5
 10           +325    ---
-----------   ----    178
That is        570
  10100011
 +11010101
-----------
 101111000
```

Multiplication and division in binary are quite simple. Except for trivial cases, they are always ‘long multiplication’ and ‘long division.’ We shall use binary division later, but no other hand-calculations are of interest.

3 Basic data types

4 For-loops and printf

5 Arithmetic in different bases

6 Integer arithmetic

(6.1) 2s complement. A short integer is 4 hex digits, 16 bits, or 2 bytes long so it can represent at most $2^{16} = 65536$ different integers. We might expect it to take values 0 to 65535, but instead half of the values are negative. The range of values is from -32768 to 32767.

Notice that 43 decimal is represented as `2b 00` hex. This shows that on our machines the first byte is *low-order*, the second is *high-order*. It is said humorously that on Intel processors, numbers are stored *little-endian*, meaning that the low-order *byte* (but not the low-order bit) is stored before the high-order byte. We should preferably write it with high-order byte first:

```
00 2b
```

This represents $2 * 16 + 11 = 43$, as expected.

Next, -9 is represented as `f7 ff`, or, high-order byte first, `ff f7`. Normally `ff ff` would represent $2^{16} - 1$ and `ff f7` would be $2^{16} - 9$. The general rules are as follows.

- Let $N = 2^{15}$. (The same idea holds for long integers, except that for long integers $N = 2^{31}$.)
- An integer x is *in short integer range* if
$$-N \leq x \leq N - 1.$$
- If $-N \leq x < N - 1$, then the *2s-complement* form of x is
$$\begin{cases} x & \text{if } 0 \leq x \leq N-1 \\ 2N + x & \text{if } -N \leq x \leq -1 \end{cases}$$
Thus a 2s-complement short integer has 'face value' between 0000 and $ffff$ (hexadecimal) or 0 and 65535 (decimal), and the signed integer it represents is in the range $-32768 \ldots 32767$.
- If x' and y' are 2s-complement integers, then their 2s-complement sum is
$$x' + y' \bmod (2N) \quad \text{i.e.,} \quad x' + y' \bmod 65536.$$
- Modular arithmetic: $x \bmod y$ is the remainder on dividing x by y. For example, $11 \bmod 4 = 3$.

(6.2) Proposition *Let x and y be two integers within the range of short integers, i.e., $-32768 \leq x, y \leq 32767$. If $x + y$ is also in this range, then 2s-complement addition will produce the correct answer in 2s-complement form.*

Partial proof. If x and y are both nonnegative, then $0 \leq x + y < 2^{15}$ and there is no carry to the 16th bit.

If x and y are both negative, and $x + y$ is in range, then $2^{16} > 2^{16} + x + y \geq 2^{15}$. But $(2^{16} + x) + (2^{16} + y) = 2^{16} + (2^{16} + x + y)$. The remainder modulo 2^{16} is $2^{16} + x + y$, and it is between 2^{15} and $2^{16} - 1$, which is correct.

Case one positive, the other nonnegative: skippped. ∎

(6.3) Converting decimal to short. Positive numbers can be converted to hexadecimal by repeatedly dividing by 16. For example, to convert 12345 to hex,

$$\begin{gathered}
12345 \div 16 = 771 \text{ remainder } 9, \quad \text{i.e.,} \quad 12345 = 16 \times 771 + 9 \\
771 \div 16 = 48 \text{ remainder } 3, \quad \text{i.e.,} \quad 771 = 16 \times 48 + 3 \\
48 \div 16 = 2 \text{ remainder } 0, \quad \text{i.e.,} \quad 48 = 16 \times 3 + 0 \\
12345 = 16 \times (16 \times (16 \times 3 + 0) + 3) + 9 \\
12345 = 16^3 \times 3 + 16^2 \times 0 + 16 \times 3 + 9 \\
(12345)_{10} = (3039)_{16}
\end{gathered}$$

To convert a negative integer x to 2s-complement, first convert $|x|$ to hex, then subtract from `ffff`, then add 1. This is the same as subtracting from $2^1 6$, as required.

For example, to convert -12345 to 2s-complement short integer,

$$\begin{gathered}(12345)_{10} = (3039)_{16}\\ ffff - 3039 = cfc6\\ cfc6 + 1 = cfc7\\ \text{Little endian:} \quad c7\ cf\end{gathered}$$

Negatives. If x is in short integer range, and so is $-x$, and the short integer representation of x is y, then $-y$ is represented as $(2^{16} - y)$, whether x is positive or negative.

The only case where x is in range, but not $-x$, is $x = -32768$, where $y = 2^{16} - y = 32768$.

(6.4) **Floating point numbers** are the computerese version of high-precision decimal numbers. They can be broken down into exponent e and mantissa m (both integers) and represent $m * 2^e$. Further details later.

(6.5) To disinguish between hex and decimal, we write, for example, $(23)_{16} = (35)_{10}$.

7 Command-line arguments (argc and *argv[])

To be able to supply your program with command-line arguments, add a bit to your main() section. First, more notation about characters and character strings.

- A single character (as opposed to a 'string' of characters) is represented with a single quote, such as

```
'a', 'A', '\n', '\0'
```

- The **null** character is represented as '\0', 8 zero-bits or `00000000`

- A character *string* is an array of characters, terminated with a null character. For example,

```
"hello\n"
```

  is stored as an array of *seven* characters, including the final null character.

- For technical reasons, a character string may be declared using a ***** rather than a **[]** notation, e.g.,

```
char * x
```

```
#include <stdio.h>

main ( int argc, char * argv[] )
{
   int i;
   for (i=0; i<argc; ++i)
     printf ( "%s\n", argv[i] );
}
```

If one compiles this program and types

```
a.out a quick brown fox
```

the four character strings `"a", "quick", "brown",` and ”fox” are called *command-line arguments.* The result is

```
a.out
a
quick
brown
fox
```

Partial explanation. You can, as shown, use to `argc` as you would use an integer variable. It means the number of character strings on the ‘command line,’ including the `a.out`. The minimum value is 1.

The variable `argv` is an *array* of *character strings*. Its size is not given, but `argv[i]` is the i-th command argument, valid for i between 0 and `argc-1`.

The command-line arguments are character strings, but they can be converted to integers, etcetera, through another `#include`:

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

If x is a character string, then

```
atoi (x)
```

is the **integer** value of x. If x does not represent an integer then `atoi(x)` is just zero.

For example,

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

```
main ( int argc, char * argv[] )
{
  int dd, mm, yy;
  dd = atoi ( argv[1] );
  mm = atoi ( argv[2] );
  yy = atoi ( argv[3] );

  printf ("Date is %02d/%02d/%02d\n", dd, mm, yy);
}
```

8 Assignment statements and stdlib.h

(8.1) Note on names of variables.

Any string of letters, digits, and/or underscores, which begins with a letter or underscore, is suitable for a variable name.

C is case sensitive, i.e., 'a' and 'A' are different. Usually one uses lowercase (small letters) in variable names. Capital letters are usually reserved for other things, though the don't have to be.

(8.2) Assignment statements usually have the form

TEMPLATE
`⟨ variable or array element ⟩ = ⟨ value ⟩ ;`

The value can be a constant, or it can be an **arithmetic expression** formed of constants, variables, etcetera. The **arithmetic operators** are

$$+, -, *, /, \%$$

the unusual one is **%** for **remainder:** $m\%n$ means the *remainder* on dividing m by n (m and n should be integers).

The old BODMAS formula of arithmetic applies here: brackets, of, division, multiplication, addition, subtraction. It is easy to make mistakes. For example, the formula for the variance of a list x_i of n numbers is

$$\frac{1}{n-1}\sum_{i=0}^{n-1}(x_i - \bar{x})^2$$

(following the C array-indexing convention; $\bar{x}$ is the mean, which needs to be calculated first). In programming this, one probably evaluates the sum first, and then the variance. Some people write

```
variance = sum / n-1;
```

This is a mistake, because it means

$$\frac{1}{n}\sum_{i=0}^{n-1}(x_i - \bar{x})^2 - 1$$

Exercise. Correct the mistake.

Left-to-right order. Generally, where the BODMAS rule does not tell you how to evaluate, left-to-right evaluation applies. For example,

```
1 + 2 + 3 = (1+2)+3 = 3+3 = 6
1 - 2 - 3 = (1-2)-3 = -1-3 = -4
1 - 2 + 3 = (1-2)+3 = -1+3 = 2
1/2/3 = (1/2)/3 = 0/3 = 0
```

There is no notation for powers. You can use things from `math.h` to compute powers.

One can compute a power using a for-loop:

```
        /* calculate x^n, assuming n >= 0 */
pow = 1;
for ( i=0; i<n; ++i )
  pow = pow * x;
```

Example using atoi(). There is a 'function' **atoi(...)** which converts character strings to 32-bit integers. It needs **stdlib.h**. Here is an example, adding up numbers from the command line

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

main( int argc, char * argv[])
{
  int x, sum, i;
  sum = 0;
  for ( i=1; i<argc; ++i )
  {
    x = atoi ( argv [ i ] );
    sum = sum + x;
  }

  printf("Total is %d\n", sum);
}
```

The remainder operator (also called the `mod` or `modulo` operator) can be used to test whether n is even:

```
if ( n % 2 == 0 ) ....
```

and it can also be used to calculate the weekday a date falls on.

(8.3) Modular arithmetic. Recall that if n and d are integers (with d positive) then there exist unique integers q and r such that

$$n = qd + r,$$
$$\text{where} \quad 0 \le r \le d - 1 :$$

q is the *quotient* and r the *remainder* on division of n by d.

Warning. In C, the quotient n/d and remainder $n\%d$ are different from the mathematical definition if $n < 0$: division 'rounds towards zero' rather than always 'rounding down.'

Modular arithmetic respects addition and multiplication. So, if

$$x + y = z$$

then

$$((x \bmod 9) + (y \bmod 9) \bmod 9) = (z \bmod 9)$$

Casting out the 9s. It is easy to calculate the remainder modulo 9. Just add digits and repeat until one digit is left.

For example,

$$12345 \mapsto 1 + 2 + 3 + 4 + 5 = 15 \mapsto 6$$

In former times, this was used to check calculations: for example, if x and y are large then there could easily be a mistake in your calculation of xy. But you can 'cast out the nines' as a check on your answer. It won't always reveal mistakes, but it usually does.

Exercise (to be discussed in lecture). Write a program converting a date to day-of-week. This is 'casting out the sevens.'

Abbreviations.

```
x = x+1;
        can be abbreviated
        (if x is an integer, long,
           or short integer)
        as
x += 1;
        or
++x;
        or
x++;

x = x-1;
        as
x -= 1;
        or
--x;
        or
x--;
```

```
x = x*y
        can be abbreviated as
x *= y;

x = x/y;
        can be abbreviated as
x /= y;
        and probably
x = x%y;
        can be abbreviated as
x %= y;
```

9 Double-precision variables

- ‘Doubles’ (double-precision floating point) occupy 8 bytes.
- Their exact layout will be discussed later.
- One can use arithmetic expressions with doubles. Where doubles and integers both occur in expressions, the integers are converted to doubles.
- Output format is

```
%f   ---  %[-]w.df  Default 6 decimal places.
%e   --- scientific notation sure of the details
%g   --- chooses whichever fits better
```

- Constants with a decimal point mean doubles. Thus $1/2$ is 0, but $1.0/2$ or $1/2.0$ all come out as $.5$. What about $1/2 + 0.0$?
- They can be read from the command line. Instead of `atoi()`, use `atof()`.
- Integer values can be assigned to double variables and vice-versa. When a double value is assigned to an integer variable, the value is *rounded towards zero.*

10 Floating-point numbers

10.1 Binary point

Like a decimal point number, a binary point number would be a number representable in the form

$$\pm a_k a_{k-1} \ldots a_0 . b_1 b_2 \ldots$$

where the a_j and the b_j are binary digits. The number represented is

$$\pm \left(\sum 2^j a_j + \sum \left(\frac{1}{2}\right)^j b_j \right)$$

There can be infinitely many digits after the 'binary point.'

For numerical work, binary floating-point numbers are much more useful. A binary floating-point number is represented as

$$\pm 2^e \; \times \; 1.b_1b_2\ldots$$

e is the **exponent**
$m = 1.b_1b_2\ldots$ the **mantissa.**
Note $1 \leq m < 2$.

Any **nonzero** real number can be represented as a binary floating-point number.

10.2 Calculating the mantissa digits.

We are given a real number m where $1 \leq m < 2$. In 'binary point,'

$$m = 1.b_1b_2b_3\ldots$$

The digits $b_1, b_2\ldots$ can be calculated as follows. Let $r_0 = m - 1$.

$$r_0 = 0.b_1b_2b_3\ldots$$

$$2r_0 = b_1.b_2b_3\ldots$$

Then b_1 is the integer part of $2r_0$. Let $r_1 = 2r_0 - b_1$:

$$r_1 = 0.b_2b_3\ldots$$
$$2r_1 = b_2.b_3\ldots$$

The integer part is b_2. Let $r_2 = 2r_1 - b_2$:

$$r_2 = 0.b_3\ldots$$

and so on. Summarising:

$$r_0 = m - 1$$

and repeatedly

$$\begin{aligned} b_{j+1} &= \text{integer part of } 2r_j \\ r_{j+1} &= \text{fractional part of } 2r_j \end{aligned}$$

Example. Calculate the floating binary representation of $80/9$. Divide by 2^3 to get

$$10/9$$

This is between 1 and 2, so the exponent is 3 and the mantissa is $10/9$.

j	r_j	$2r_j$	b_{j+1}
0	$\frac{1}{9}$	$\frac{2}{9}$	0
1	$\frac{2}{9}$	$\frac{4}{9}$	0
2	$\frac{4}{9}$	$\frac{8}{9}$	0
3	$\frac{8}{9}$	$\frac{16}{9}$	1
4	$\frac{7}{9}$	$\frac{14}{9}$	1
5	$\frac{5}{9}$	$\frac{10}{9}$	1
6	$\frac{1}{9}$	$\frac{2}{9}$	0

This is a point of recurrence, and the pattern will repeat. Therefore

$$\frac{10}{9} = 1.000111000111000111\ldots$$

It can be checked by summing a geometric series

$$\begin{gathered}1 + \frac{7}{64} + \frac{7}{64^2}\cdots \\ = 1 + \frac{7}{64}\left(\sum_{j\geq 0}\frac{1}{64^j}\right) \\ = 1 + \frac{7}{64}\left(\frac{1}{1-1/64}\right) = \\ 1 + \frac{7}{64}\frac{64}{63} = \\ 1 + 1/9 = 10/9.\end{gathered}$$

Thus the binary floating-point representation of $80/9$ is

$$2^3 \times 1.000111000111000111\ldots$$

10.3 Scientific notation.

Integer arithmetic is not much use for scientific computation which needs *accuracy.* Hence, single- and double-precision floating point numbers.

Scientific notation. In scientific applications, ordinary decimal numbers are too long for convenience, so a notation of the form

±⟨mantissa⟩E⟨exponent⟩

is used. For example, the number

$$123000000000$$

is more compactly represented as

$$1.23E + 9$$

The ‘correct’ position of the decimal point depends on the exponent; the decimal point is ‘floating’; hence the term floating point. A (decimal) floating-point number is *normalised* if its mantissa is at least 1 and less than 10.

According to the IEEE floating-point standard, a (single-precision) floating-point number occupies 4 bytes. It consists of *sign, mantissa,* and *exponent.* It is *normalised* if its mantissa is at least 1 and less than 2.

- In a single-precision floating-point number, the high-order bit[3] is the *sign bit*, 1 for negative, 0 for positive. The sign is separate, i.e., the mantissa is not 2s complement.
- The exponent E is stored in the next 8 bits, ‘biased,’ not 2s complement: 00_{16} means -127 and ff_{16} means $+128$. In other words, $E + 127$ is stored in 8 bits, assuming it fits.
- The **fractional part of the** mantissa is stored in the 23 low-order bits.
- Zero is an exception, represented as a string of 32 zero-bits.

Double-precision floating-point numbers occupy 8 bytes. They use the same ideas as in single-precision, but now they have

- a sign bit
- 11 exponent bits, and
- the 52 low-order bits of the mantissa.

(10.1) Why use this representation? Because accurate calculation should *scale up and down.* Suppose you measure the circumference of a sphere accurate to the nearest inch. Is that accurate or not? Depends on the sphere.

$$\boxed{\textit{Accuracy is measured as a proportion.}}$$

With floating-point numbers, there must be loss of accuracy sometimes. However, the IEEE standard makes certain **guarantees** which we shall call *the IEEE promise.*

Example. Convert $80/9$ to single-precision floating-point.

- The sign bit is zero.
- The exponent is adjusted so that the mantissa is between 1 and 2: $8 \times (10/9)$. The ‘true’ exponent is 3 and the mantissa should approximate $10/9$. The infinite precision floating point representation has been computed in a previous example.

Now we must continue the fractional part until it exceeds 23 bits.

$$\boxed{.00011100011100011100011}\boxed{1000111\ldots}$$

The part after the 23rd bit is more than half the value of that bit, so rounding is upwards, — meaning 1 is added to the 23-bit string as if it were a 23-bit face-value integer.

$$\boxed{.00011100011100011100100}$$

Now we can assemble the floating-point representation.

[3]Floating-point numbers are ‘little-endian’ on Intel processors. The high-order bit is the high-order bit of the fourth byte. We stick to ‘big-endian’ descriptions.

- sign bit 0
- True exponent 3, biased exponent $127+3 = 130$. This is 10000010 in 8-bit (face-value) binary.
- The mantissa is shown above.

Hence the number in binary is

```
0 1000 0010 000111 000111 000111 00100 =
0100 0001 0000 1110 0011 1000 1110 0100 =
   4    1    0    e    3    8    e    4
little endian: e4 38 0e 41
```

Example. $-5/1152 = -5/(9 \times 128)$.

- Sign bit 1.
- Normalise: mantissa becomes $10/9$, same as before.
- Exponent:
  $$-5/1152 = -\frac{10/9}{256}$$
  and $256 = 2^8$, so the exponent is -8.
- Add 127.
  $127 - 8 = 119 = 64 + 32 + 16 + 4 + 2 + 1$, so the exponent is represented as $0111\ 0111$.

```
1 0111 0111 000111 000111 000111 00100
1011 1011 1000 1110 0011 1000 1110 0100
   b    b    8    e    3    8    e    4

little endian: e4 38 8c bb
```

10.4 Double precision

The double-precision layout is, briefly,

$$\boxed{\textbf{1+11+52, bias 1023}}$$

(and little endian).

Example. $-5/1152 = -5/(9 \times 128)$.

- Sign bit 1.
- Normalise: mantissa becomes $10/9$, same as before.

- Exponent:
  $$-5/1152 = -\frac{10/9}{256}$$
  and $256 = 2^8$, so the exponent is -8.
- It is easier to compute the biased exponent directly in binary

```
   0 1111111111
  -        1000
   0 1111110111
```

- `000111 000111 000111 000111 000111 000111 000111 000111 0001 : 11 000`

  Round up

  ```
  000111 000111 000111 000111 000111 000111 000111 000111 0010
  ```

Putting together

```
1 01111110111 000111 000111 000111 000111 000111 000111 000111 000111 001
1011111101110001110001110001110001110001110001110001110001110010
1011 1111 0111 0001 1100 0111 0001 1100
   b    f    7    1    c    7    1    c
0111 0001 1100 0111 0001 1100 0111 0010
   7    1    c    7    1    c    7    2

Little endian 72 1c c7 71 1c c7 71 bf
```

11 Scanf() and input/output redirection

(11.1) The easiest way to supply a little input to your program is through command-line arguments.

Another way to get data is through the `scanf` procedure. Using `scanf`, one can read data from the terminal. `Scanf` is intended to be a kind of opposite to `printf,` in the sense that what `printf` prints below should be what `scanf` expects below.

```
printf("m=%d\n",m);
scanf("m=%d",m); /* WARNING: THIS IS WRONG */
```

There is an important difference here. We have used `printf` to print arithmetic expressions and strings. For example,

```
printf ( "m=%d, 2*m=%d\n", m, 2*m );
```

We do not expect to ‘scan’ `2*m`. While `printf` **prints expressions**, `scanf` **reads variables.** More exactly,

> **Printf** prints **values** to the terminal. **Scanf** reads data from the terminal and stores it at variables’ **addresses**.

(11.2) Definition *Given a variable* `x`*, its* address *is given by the expression* `&x`*. This must be used for the basic data types* `char, int,` *etcetera, though not for character strings.*

Character strings are different. They are arrays, and will be studied later.
We shall use `scanf` to read numeric data only. It would be confusing to do more with it.
Rules for interpreting the format control string:

- ‘White space’ — blanks, tabs, and newlines, are **generally** ignored (but newlines are more complicated).
- `scanf` reads input and *returns a value*, namely, the number of items successfully matched.

Keep things simple. Only use `scanf` for reading a list of numbers from the terminal, without any fancy formatting.
Here’s an example of scanning using a for-loop.

```
#include <stdio.h>

main()
{
  int i, m;
  int a[10];

  scanf ( "%d", &m ); /* m = number of items,
                       * assumed at most 10 */

  for ( i=0; i<m; ++i )
    scanf ( "%d", & ( a[i] ) );

  printf ( "%d items in array\n", m );
  for ( i=0; i<m; ++i )
    printf ( "%d ", a[i] );
  printf("\n");
}
```

Notice that data begins *after* the command line.

```
% gcc scan1.c
% a.out
3
```

```
1
2
3
3 items in array
1 2 3
CTRL-D
%
```

Here's a version which uses scanf to detect END-OF-DATA, so there is no need for the number of items input.

```
#include <stdio.h>

main()
{
  double val, x[1000];
  int num_read;
  int n;

  num_read = 1;

  n = 0;
  while ( num_read == 1 )
  {
    num_read = scanf ( "%lf", &val );
    if ( num_read == 1 )
    {
      x[n] = val;
      ++n;
    }
  }
  printf("n=%d\n", n);
}
```

NO NEED to be so complicated. Here's a simpler version

```
#include <stdio.h>

main()
{
  double val, x[1000];
  int num_read;
  int n;

  n = 0;
  while ( scanf ( "%lf", &val ) == 1 )
  {
    x[n] = val;
    ++n;
  }
  printf("n=%d\n", n);
}
```

```
%gcc scan2.c
%a.out
1 2
3
3 items in array
1 2 3
CTRL-D
%
```

END-OF-DATA is signalled by CTRL-D

Input redirection. One can prepare a file, called mydata, say, containing, say

```
1
2  2
```

and type

```
a.out < mydata
```

One gets the same results as above.

NOTE: *there is no need to include CTRL-D in the file 'mydata.'*

Scanf() and doubles. *Be very careful about the format when inputting doubles using scanf().*

```
  double x;
  scanf ( "%f", &x );
```

is **wrong.** It would be correct for inputting `floats`. Use

```
  scanf ( "%lf", & x );
```

The ’l’ is the letter ell, and ‘lf’ means ‘long float.’ What

```
  double x;
  scanf ( "%f", &x );
```

does is to convert the input number to a single-precision float and store it in the first (low-order) four bytes in x. The other four bytes are left uninitialised.

12 ‘While’ loops

A while-loop is related to a for-loop. Actually, one could write every while-loop as a for-loop, but it could be artificial. The general form is

```
while ( <condition holds> )
  <statement; or {group}>
```

Using a while-loop in conjunction with `scanf`, one can process a long list of numbers

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

main()
{
  int i, x, sum, count;
  sum = 0;
  count = 0;
  while ( scanf ( "%d", & x ) == 1 )
  {
    sum += x;
    ++ count;
  }
  printf("Sum of %d numbers is %d\n", count, sum);
}
```

Sample run (the above program is in `s.c`)

```
% gcc s.c
% a.out
3 4 5
1 6
CTRL-D
Sum of 5 numbers is 19
%
```

Note how to signal end-of-data:

$$\boxed{\textbf{CTRL-D for end-of-data}}$$

(12.1) More about while-loops. For example, here is an inefficient way to compute quotient and remainder using a while-loop.

```
/* piece of code: quotient and remainder dividing n by d,
 * assuming d > 0
 */

  q = 0; r = n;

  while ( r >= d )
  {
    q = q+1;
    r = r-d;
  }
```

Let us simulate this code with $d = 13$ and $n = 100$.

```
    q      r     r>=d?
    0    100        1
    1     87        1
    2     74        1
    3     61        1
    4     48        1
    5     35        1
    6     22        1
    7      9        0
```

```
quotient 7, remainder 9.
```

Example. Suppose m and n are integers, and $m \geq n > 0$.

```
  int x,y,z; ....

  x = m; y = n;

  while ( y != 0 )
  {
    z = x % y;
    x = y;
    y = z;
  }
```

Try $m = 165, n = 24$.

```
  x        y        z
165       24       21
 24       21        3
 21        3        0
  3        0
```

13 If-statements and conditions

(13.1) Conditions and if-statements. An if-statement has the form (mind the **INDENTATION**)

```
if ( <condition> )
  <statement; or {group}>

and --- optionally ---
else
  <statement; or {group}>
```

The condition must be in parentheses.

$\boxed{\textbf{if (<condition>) \ldots}}$

Programming languages usually use the word 'then.' C doesn't. The condition is in parentheses and 'then' is understood.

Statement or group of statements? It is best practice to use curly brackets *always*, as otherwise one gets into a mess. (If I forget to do so, remind me.)

```
if ( x == 1 )
{
  printf ("hello\n");
}
else
{
  printf ("goodbye\n");
}
```

Conditions are converted to integers. In **a.out** the condition `argc == 2` is tested and an integer produced: 1 for true and 0 for false. More generally, any integer value can be used as a condition; nonzero is treated as true and zero as false.

Complex if-statements. The basic 'if-statement' relations are

```
==, <, <=, >, >=, !=
```

They can be grouped into more complex statements using

```
&& for ‘and,’
|| for ‘or,’ and
! for ’not.’
```

For example, to test if a 4-digit year is a leap-year,

```
if ( yy % 400 == 0 || ( yy % 4 == 0 && yy % 100 != 0 ) )
```

Every fourth year is a leap year, except for centuries; every fourth century is a leap year.

More complex conditions can be constructed with

$$\boxed{\textbf{\&\&} \quad || \quad \textbf{!}}$$

for **and, or, not**. The **DOUBLE** ampersand and double bar are important; single ampersand and single bar have a different meaning.

For example, suppose yy represents a year, including the century, not just the last two digits. According to the Gregorian calendar, a leap year is

- divisible by 4, **and**
- **either is not** divisible by 100 **or is** divisible by 400.

Meaning that only one century in 4 is a leap-year; so on average the year is

$$365\frac{397}{400}$$

days long, apparently a good approximation.

This can be expressed in C:

```
... int leapyear, yy; ....

  leapyear =
    yy % 4 == 0   &&
    ( yy % 100 != 0 || yy % 400 == 0 )
  ;

  if ( leapyear ) ....
```

Why the parentheses? Because I don’t remember the evaluation rules (an extension of the rules for arithmetic operations).

14 Nested if-statements and indentation

There is an ambiguity when `if`-statements are combined.

```
if (A)
  if (B)
    C;
  else
    D;
```

In this example, the `else` can be interpreted as either being when A is true and B false, or when A is false. The former is correct. Thus the above is equivalent to

```
  if (A)
  {
    if (B)
      C;
    else
      D;
  }
```

and if you want the other, you must use braces

```
  if (A)
  {
    if (B)
      C;
  }
  else
    D;
```

Frequently one applies tests `A,C...` in succession.

```
  if (A)
    B;
  else
    if (C)
      D;
    else
      if (E)
        F;
      else
        G;
```

If A is true, do B. If A is false and C true, do D. If A and C are false and E is true, do F. Else do G.

To avoid excessive indentation, I prefer not to indent in this case:

```
  if (A)
    B;
  else if (C)
    D;
  else if (E)
    F;
  else
    G;
```

15 Array initialisation

In C, it is possible to ‘initialise’ variables when they are declared.

It is not always a good idea to initialise variables in this way. It is most useful for creating tables of data.

For example, one can create an array giving the number of days in each month (not a leap-year) as follows

```
int month_length[12]
  = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

The curly braces **and the semicolon** are required.

In fact, when array initialisation is used, it is not necessary to give the size of the array:

```
int month_length[]
  = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

This is dangerous because the array size is not immediately obvious.

Similarly

```
char hello[] = {’H’,’e’,’l’,’l’,’o’,’\0’};
```

defines a character string ”Hello”. One can also write

```
char hello[] = "Hello";
or
char hello[6] = "Hello";
```

Again, one can define abbreviated names for the days of the week:

```
char * weekday[7] = {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};
```

This notation is like the `*argv[]` notation.
In class we shall write a program converting date to day of week.

16 Day of week program

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

main( int argc, char *argv[] )
{
  int dd, mm, yy, leapyear, century_adjustment, result;

  char * weekday [7] =
    {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};

  char * month[12] =
    {"January","February","March","April","May","June",
   "July","August","September","October","November","December"};


  int month_offset[12] =
  {0,3,3,6,1,4,6,2,5,0,3,5};

  century_adjustment = 6; /* for 21st century */

  dd = atoi ( argv [1] );
  mm = atoi ( argv [2] );
  yy = atoi ( argv [3] );

        /*
         * ASSUMED: 0 <= yy <= 99, simplifying
         * the calculations
         */

  leapyear = ( yy % 4 == 0 );

  result = dd + month_offset[ mm-1 ] + yy + yy/4 + century_adjustment;
  if (leapyear && mm <= 2 )
    result = result + 6;
  result = result % 7;

  printf("%s, %d %s %02d\n", weekday[ result ], dd,
        month[ mm-1 ], yy);
}
```

17 2-dimensional arrays

Matrices should be represented using 2-dimensional arrays. C allows this. The declaration should be (for an array of doubles)

```
                double a [3][4];
```

defines a 3×4 matrix.

We would think of the above array as consisting of 3 rows, each row containing 4 elements. C treats a 2-dimensional array `a[3][4]` as an array of 3 1-dimensional arrays, each containing 4 elements. The entire array is kept in a single block of memory (as usual, `a` contains the address where the block begins).

Size of a 2-dimensional array. For example, `double a[3][4]`. It consists of 3 arrays each containing 4 doubles: 3 blocks of $4 \times 8 = 32$ bytes. The size is $3 \times 32 = 96$.

Indexing. General array element is indexed as `a[i][j]`, where $0 \leq i <$ (no. of rows) and $0 \leq j <$ (no. of columns).

The address of `a[i][j]` is

```
starting address of a +
  i * ( size of a row ) +
    j * ( size of each entry )
=
starting address of a +
  ( i * (no. of columns) + j ) * ( size of each entry )

For example, given
  double a[4][5];
Suppose a starts at address 1000.  Then
  a[2][3] starts at
1000 + ( 2 * 5 + 3 ) * 8
  = 1104.
```

Generally, `<name> [<rows>][<columns>]`. The *width* is the number of *columns*, and the *height* is the number of *rows*. Hence

⟨name⟩ [⟨rows⟩][⟨columns⟩]
⟨name⟩ [⟨height⟩][⟨width⟩]

Example. Program to read

$$m \;\; n \;\; a_{00} \ldots a_{m-1,n-1}$$

into an array, and print it.

Printf format 'g' This code uses 'g' format rather than 'f.' The 'g' format allows scientific notation to be used where necessary, but generally produces the simplest output.

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

main (int argc, char * argv[] )
{
  int i,j,m,n;
  double a[10][10];

  scanf("%d %d", &m, &n );

  if ( m > 10 || n > 10 )
  {
    printf("%s: dimensions %d %d too large\n",
      argv[0], m, n);
    exit (-1);
  }

  for ( i=0; i<m; ++i  )
    for ( j=0; j<n; ++j )
      scanf ("%lf", &(a[i][j]) );


  printf("%d %d\n", m, n);
  for ( i=0; i<m; ++i )
  {
    for ( j=0; j<n; ++j )
      printf("%8g", a[i][j]);
    printf("\n");
  }
}
```

Multiply column vector by matrix on left. The following routine does this.

```
#include <stdio.h>

main()
{
  double a[10][10], x[10], y[10], sum;
  int m,n;
  int i,j;

  scanf("%d %d\n", &m, &n);
  for ( i=0; i<m; ++i )
  {
    for (j=0; j<n; ++j )
    {
      scanf("%lf", &(a[i][j]));
    }
  }

  for ( j=0; j<n; ++j )
  {
    scanf("%lf", &x[j]);
  }

  printf("Ax: ");

  for ( i=0; i<m; ++i )
  {
    sum = 0;
    for ( j=0; j<n; ++j )
    {
      sum = sum + a[i][j] * x[j];
    }
    y[i] = sum;
  }

  for (j=0; j<n; ++j )
    printf(" %6g", y[j]);
  printf("\n");
}

data file d:
3 3
1 2 3
4 5 6
7 8 9
3 -2 1

a.out < d:
Ax:      2      8     14
```

18 Functions and subroutines

A C program has the following general structure

```
#include etcetera


main ( with or without command-line arguments )
{
  declare all 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
%
```

- 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:

```
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 );
}
```

A last 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 );
}
```

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.

```
int A ( int n );

void B ( )
{
  int x;
  x =  A ( 5 );
  ....
}

main()
```

```
{
  B();
}
```

TEMPLATE for a function or routine
In routines, `<type>` is `void`
In functions, the calculations include
`return` statements

```
<type>  <name>          (<arguments>)
{
        <variables>;
        <calculations>;
}
```

19 Simulating subroutines and functions

Simple routines are easiest to understand by tracing (i.e., simulating) their action on a simple piece of data. One makes a table giving the values of all variables, one column for each variable, and enters values in the order in which they are produced by the program. For example, trace the following program and say what the routine does in general.

```
#include <stdio.h>

int xxx ( int n )
{
  int i,x;
  x = 1;

  for (i=0; i<n; ++i )
  {
    x = x + x;
  }
  return x;
}

main()
{
  printf("xxx(5)==%d\n", xxx(5));
}
```

Simulation:

```
   i       x       n
                   5
           1
   0
           2
   1
           4
   2
           8
   3
          16
   4
          32
   5
   returns 32
Prints
xxx(5)==32
```

Clearly the function returns 2^n. Another example

```
#include <stdio.h>

int yyy ( int n )
{
  int i, x, s;
  s = 0;
  x = 1;
  for ( i=0; i<n; ++i)
  {
    s = s+x;
    x = x+2;
  }
  return s;
}
main()
{
  printf("yyy(5)==%d\n", yyy(5));
}
```

```
   i   x   s   n
               5
           0
       1
   0
           1
```

```
        3
    1
            4
        5
    2
            9
        7
    3
           16
        9
    4
           25
       11
    5

    returns 25
prints
yyy(5)== 25
```

'Clearly' the function returns n^2. (Summing the odd integers produces the perfect squares.)

20 Gauss-Jordan elimination

This is not examinable. Gauss-Jordan elimination applies operations `swap, scale, subtract` to the rows of an array to bring it into so-called *reduced row-echelon form.* If the matrix is an augmented matrix representing a system of linear equations, the set of all solutions can be derived from the reduced matrix. If the matrix is an $n \times 2n$ matrix consisting of an $n \times n$ matrix A followed by the $n \times n$ identity matrix: schematically

$$A||I$$

then if A is invertible, the reduced matrix will be

$$I||A^{-1}.$$

If, in the reduced matrix, the left-hand side of the bottom row is zero, then A is not invertible.

```
#include <stdio.h>

void swap( int i, int k, double a[][20] )
{
  double x;
  int j;

  for ( j=0; j<20; ++j )
  {
    x = a[i][j];
    a[i][j] = a[k][j];
    a[k][j] = x;
  }
}

void scale ( int k, double a[][20], double c )
{
  int j;
  double x;

  for (j=0; j<20; ++j)
    a[k][j] = c * a[k][j];
}

void subtract_multiple ( int k, double a[][20], double c, int i )
{
  int j;
  double x;

  for ( j=0; j<20; ++j )
    a[i][j] = a[i][j] - c * a[k][j];
}

void reduce ( int m, int n, double a[][20] )
{
  int i,j,k,r;
  k = 0;

  for ( j=0; j<n; j = j+1 )
  {
    r = -1;
    for ( i=k; i<m && r<0; ++i )
      if (a[i][j] != 0)
        r = i;

    if ( r >= 0 )
    {
      swap ( k, r, a );
      scale (k, a, 1/a[k][j]);
      for (i=0; i<m; ++i)
        if ( i != k )
          subtract_multiple(k, a, a[i][j], i);
      ++k;
    }
  }
}

main()
{
  int i,j,m,n;
  double a[10][20];

  scanf("%d %d", &m, &n);
  for (i=0; i<m; ++i)
    for (j=0; j<n; ++j )
      scanf("%lf", &( a[i][j] ));

  reduce (m,n,a);

  printf("Reduced matrix\n");
  printf("%d %d\n", m, n);

  for  (i=0; i<m; ++i)
  {
    for (j=0; j<n; ++j)
      printf(" %8.5g", a[i][j]);
    printf("\n");
  }
}
```

```
Sample input

3 6
1 2  3 1 0 0
2 3  1 0 1 0
1 1 -1 0 0 1

Reduced matrix
3 6
        1        0        0        4       -5        7
       -0        1        0       -3        4       -5
        0        0        1        1       -1        1

Another input (matrix not invertible)

3 6
1 2 3 1 0 0
4 5 6 0 1 0
```

```
7 8 9 0 0 1

Reduced matrix
3 6
        1         0        -1         0  -2.6667   1.6667
       -0         1         2         0   2.3333  -1.3333
        0         0         0         1       -2        1
```

21 Gaussian elimination

We are introducing Gaussian elimination with partial pivoting for the sake of variety — as an alternative to Gauss-Jordan elimination.

The idea is to illustrate how one writes linear algebra code in C.
Specialists in numerical linear algebra do not favour Gauss-Jordan elimination.
The difference between the two procedures is as follows.

- Gauss-Jordan elimination applies to matrices of any size, bringing them to reduced row-echelon form.
- When applied to the augmented matrix for a set of n independent linear equations, Gauss-Jordan elimination reduces the augmented matrix fully, leaving the identity matrix on the left and the solution in the rightmost column. Schematically,

  $$A||B \quad \mapsto \quad I||X.$$

  where $X = A^{-1}B$.
- In Gauss-Jordan elimination, the rows are scaled so the diagonal elements are 1.
- Gaussian elimination, applied to the augmented matrix for a system of n independent equations, brings the first n columns into upper triangular form. Schematically

  $$A||B \quad \mapsto \quad U||Y$$

  and the solution $X = A^{-1}B = U^{-1}Y$ is calculated by *back substitution.*
- Gaussian elimination has a policy of *partial pivoting* which swaps the rows around to improve accuracy.

  The idea is keep fairly large numbers (large in absolute value) on the diagonal, to avoid inflating errors by dividing by small numbers.
- In Gaussian elimination, the diagonal elements are not scaled.

Thus, Gauss-Jordan elimination would reduce

$$\begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 8 \end{bmatrix}$$

to

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix},$$

Gaussian elimination would produce

$$\begin{bmatrix} 3 & 6 & 9 \\ 0 & 1 & 2 \end{bmatrix}$$

The answer is not stored in the rightmost column, but can be calculated by *back substitution*. That is,

$$y = 2 \quad \text{from bottom row}$$
$$3x + 12 = 9, \quad x = -3/3 = -1$$

With Gaussian elimination, the diagonal entries are not scaled to make them 1. **Partial pivoting** means ensuring that the diagonal entries are as large as possible in absolute value.

For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \end{bmatrix}$$

Since $2 > 1$, the rows are swapped.

$$\begin{bmatrix} 2 & 5 & 8 \\ 1 & 2 & 3 \end{bmatrix}$$

and we finish with

$$\begin{bmatrix} 2 & 5 & 8 \\ 0 & -1/2 & -1 \end{bmatrix}$$

Here is another example. Solve $AX = B$, where

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} \quad \text{and } B \text{ is } \begin{bmatrix} 2 \\ 5 \\ 9 \end{bmatrix}.$$

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

7    8   10   9  clear column
4    5    6   5
1    2    3   2

7    8   10   9
0    3/7 2/7  -1/7 pivot
0    6/7 11/7 5/7

7    8   10   9
0    6/7 11/7 5/7
```

```
  0    3/7 2/7  -1/7

  7    8   10   9
  0    6/7 11/7 5/7
  0      0 -1/2 -1/2

back substitution
             z = 1
     6y/7 + 11/7 = 5/7
      y       = -1
  7x  - 8  + 10  = 9
   x          = 1
```

You will be asked to apply back-substitution in the next programming assignment. That is, you will be writing a routine

```
void back_sub ( int n, double u[10][10], double y[10], double x[10] )
{ ... }
```

where u is upper triangular and you solve $ux = y$.

The formula is applied for $i = n-1, \ldots, 0$, in descending order. Since u is upper triangular, $u_{ij} = 0$ if $j < i$, so

$$\sum_{j=0}^{n-1} u_{ij} x_j = b_i$$

becomes

$$\sum_{j=i}^{n-1} u_{ij} x_j = b_i$$

or, in other words,

$$x_i = \frac{b_i - \sum_{j=i+1}^{n-1} u_{ij} x_j}{u_{ii}}$$

22 C has call-by-value

Consider

```
#include <stdio.h>
void print ( int n )
{
  printf("Integer... %d\n", n);
  ++n;
  printf("Integer... %d\n", n);
}
```

```
main()
{
  int m = 3;
  print(m);
  print(m);
}
```

the routine argument `n` is like an *initialised local variable.* Compiling and running the program:

```
%gcc p.c
%a.out
Integer... 3
Integer... 4
Integer... 3
Integer... 4
%
```

That is, the value of `m` is copied to `n,` and `n` is local to the routine. The change to `n` (local) does not affect `m` (in the calling routine `main`). If we change the `main()` routine to

```
main()
{
  print (3);
  print (3);
}
```

we get the same output.

There are 5 recognised ways of argument-passing (parameter-passing)

- Call by value
- Call by reference
- Call by result
- Call by value-result
- Call by name

The last three are irrelevant to us: the last is bizarre, occurring in the 1960s language Algol and in the 'funarg problem' in Lisp.

Actually, the `#define` feature in C, which should **never** be used except to give names to constants, if used with arguments has all the difficulties of call-by-name.

In call-by-reference, the subroutine argument is identical with the argument passed, i.e., occupies the same memory location. This was the natural way when Fortran was invented, and in the early compilers it had very odd effects.

Call-by-reference is easily simulated in C using pointers. We shall revise the program to simulate call-by-reference.

```
#include <stdio.h>
void print ( int * n )
{
  printf("Integer... %d\n", * n);
  ++ ( * n );
  printf("Integer... %d\n", * n);
}

main()
{
  int m = 3;
  print( &m );
  print( &m );
}
```

Running it,

```
%a.out
Integer... 3
Integer... 4
Integer... 4
Integer... 5
%
```

In most (or all?) languages, the constant 3 would be stored in a memory location when the program `a.out` was loaded into central memory, and whenever 3 was used in the program, the value would be taken from this location. In some early Fortran compilers, the following could happen — illustrated as if it would happen in C, that is, if C had *call-by-reference*:

```
#include <stdio.h>
void print ( int n )
{
  printf("Integer... %d\n", n);
  ++n;
  printf("Integer... %d\n", n);
}

main()
{
  print( 3 );
  print( 3 );
}
```

```
%a.out
Integer... 3
Integer... 4
Integer... 4
Integer... 5
%
```

The programming language PL/I had a mixture of call-by-reference together with 'automatic conversion' which made odd things happen. If you weren't very careful, subroutine calls would be call-by-reference sometimes and call-by-value other times.

Summarising

C has call-by-value. Subroutine arguments are initialised local variables.

Exercise. gcc compiles the following program, but with warnings.

```
#include <stdio.h>
void print ( int * n )
{
  printf("Integer... %d\n", * n);
  ++ ( * n );
  printf("Integer... %d\n", * n);
}

main()
{
  print( 3 );
  print( 3 );
}
```

Run it, and try to explain what happens.

23 Subroutine array arguments

An array can be passed to a subroutine or function without declaring its size.

```
void negative ( int x[], int count )
{
  int i;
  for ( i=0; i<count; ++i )
    x[i] = - x[i];
}
```

```
main()
{
  int a[3] = {1,2,3};
  int i;

  negative ( a, 3 );
  for (i=0; i<3; ++i )
    printf(" %d", a[i]);
  printf("\n");
}
```

- It is necessary to pass `count` to the routine, since the size of the array is not otherwise available.
- The value of `a` is passed to the routine.
- This value is the starting address of array `a`.
- Therefore, in the routine, the value of `x` is where is where the array `a` begins.
- `x[i]` is the integer stored i (integer) places beyond the start of array `a`.
- In other words, every reference to `x[i]` in the routine actually refers to `a[i]`.

In other words:

Array arguments are effectively call-by-reference.

Remarks.

- This is very economical, since it avoids copying large arrays.
- The argument `x` can also be defined simply as a pointer:

  ```
  void negative ( int * x, int count )
  ```

  would work just as well, and is common practice.
- Usually a subroutine argument of type `char *` is intended for character strings:

  ```
  void reverse_string ( char * s )
  ```

  `Count` is not needed, because end-of-string is marked by `'\0'`.

24 Global variables

Data is normally passed to subroutines and functions by passing arguments. Sometimes it is cleaner to use *global variables,* which are 'visible' to routines and functions.

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

double a[10][10], x[10], y[10];

int matvecprod ( int m, int n)
{ ...  }

main()
{ ...  }
```

Because `a,x,y` have been declared above the routines `matvecprod()` and `main()`, they are 'visible' to both and it is not necessary to pass them as arguments. The same was not done with `m,` `n`; it could have been.

This is an example of **global** variables. The variables declared within the routines are **local** to those routines. They cannot be seen or modified from outside these routines. This is true also for the `main()` routine.

Data may be communicated through **global variables.**

```
#include ....

char * day[7] = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

int aaa ( ... )
{ int n=0; ...  }

...

main ( ... )
{ int n=1; ...  }
```

- Both `aaa` and `main` can use the array `day[]`; it is called a *global variable.*
- The variables `n` in `aaa` and `main` are *local variables,* and are independent of each other.
- This gives 3 kinds of variable.
    - global,
    - local, and

    - routine argument.

- A local variable can be initialised as shown. It is initialised each time the routine is called: once for `main`, perhaps often for `aaa`.

- The global variable `day[]` is initialised once, of course.

The global variables are *static*, meaning that they 'last' as long as the program is running. The local variables 'last' as long as the routine is running. When the routine ends, their value is lost, and if the routine begins again, the variables are new, whether or not they are initialised.

This will be explained fully later.

There is another possibility. It is possible for a routine variable to be declared 'static.'

```
void aaa ( ... )
{
  static int x = 0;
}
```

The variable x *does* last throughout the program. It is initialised to 0. It has the *lifetime* of a global variable and the *scope* of a local variable, meaning that it is invisible outside the routine `aaa`.

Summary

	Scope	Lifetime	Initialised how often?
local	private to routine	lasts for a single run of the routine	initialised (if at all) at start of every run of the routine
global	accessible from all routines	lasts for entire run of the program	initialised once
routine argument	private to routine	single run	start of every run
static local variable	private to routine	lasts entire program	intialised once

25 String processing

When `char *` is used as a type, character strings are almost always intended, that is, arrays in which `'\0'` indicates end of string.

Here are some useful functions.

- **int strlen (char * s)** in **string.h** returns the length of s

- **int strcomp (char * x, char * y)** compares two strings. It is used to sort lines of text. It behaves in the following peculiar way.

    - If x comes before y in lexicographical (dictionary) order, then `strcomp()` returns a negative number. Almost any negative number is possible.

  - If x and y are equal as strings (they have the same length), then `strcmp()` returns 0.
  - If x comes after y in lexicographical (dictionary) order, then `strcomp()` returns a positive number. Almost any positive number is possible.

  Mnemonic: it is as if `strcmp()` returns $x - y$.

  This is also in `string.h`

- **char * fgets (char * buffer, int len, FILE * file)** reads up to end-of-line (including `'\n'`), or end-of-data, or up to `len` characters ended with `'\0'`, whichever comes first.

  This prevents characters being stored beyond the end of the buffer (‘buffer overflow’).

  This is in `stdio.h`.

- **snprintf (char * buffer, int len, char * format, item . . .)** formats the items for printing like `printf()`, but formats them into the string `buffer` rather than printing to terminal.

  Again, `len` is there to prevent buffer overflow.

  This is in `stdio.h`.

When processing text, the following routine is helpful without being absolutely necessary. It removes the newline from a string.

This is useful because `fgets()` usually includes newlines, but not always. It’s best to make sure all newlines are deleted.

```
#include <string.h>
void delete_newline ( char * s )
{
  int last = strlen ( s ) - 1;
        /* last character before '\0' */
  if ( last >= 0 && s[last] == '\n' )
    s[last] = '\0';
}
```

Here is another method, which produces the same result so long as the newline can only be at the end of the string.

```
void delete_newline ( char * s )
{
  int i;
  for ( i=0; s[i] != '\0'; ++i )
  {
    if ( s[i] == '\n' )
    {
      s[i] = '\0';
      return;
    }
  }
}
```

Here is a long, almost useful, example. It reads lines of text, breaking them into words (nonblank strings separated by blanks).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
        /* ctype.h: for isspace(), etcetera */

void del_nl ( char * s )
{
        /*
         * delete newline
         */
  int i;
  for ( i=0; s[i] != '\0'; ++i )
  {
    if ( s[i] == '\n' )
    {
      s[i] = '\0';
      return;
    }
  }
}
```

The processing is done within `main()`. First, here is an *outline* of what `main()` does; parts are left out, with comments to explain them.

```
main()
{
        /*
         * get max_len off the command line
         */
  max_len = atoi ( argv[1] );
```

(25.1) The outermost while-loop controls reading of text.

```
  while ( fgets( buffer, 200, stdin ) != NULL )
  {
    i = 0;
```

(25.2) The next-outermost loop controls reading of the buffer. Variable i will index the beginning of words,

```
    while ( buffer[i] != '\0' )
    {
        /*
         * increment i until a non-space character is
         * found, or end-of-string
         */

        /*
         * if not end-of-string, then
         * another word begins at i
         */
```

(25.3) The if-statement below controls processing of the next word in the buffer.

```
      if ( buffer[i] != '\0' )
      {
          /*
           * copy word to word buffer. Stops when j
           * is at a blank or end-of-string
           */

        while ( buffer[j] != '\0' && ! isspace ( buffer[j] ) )
        {
        }

          /*
           * Calculate the potential output line length
```

```
         * if the word is printed on same line
         */

        /*
         * If not too long, print word on same line,
         * else start a new line. No provision is
         * made for breaking long words, so lines
         * longer than max_len are still possible.
         */

    }
```

This ends the part controlled by the if-statement introduced in §25.3.

```
    } /* look for next word */
```

This ends the part controlled by the while-loop introduced in §25.2.

```
  } /* read next line  */
        /* print an extra newline */
  printf("\n");
}
```

This ends the outermost while-loop (§25.1) and the main routine. That is an outline of the main program: the actual main program is below.

```
main( int argc, char * argv[])
{
  char buffer[200], word_buffer[200];
  int i,j,k, max_len, line_len, word_len, new_len;

        /*
         * get max_len off the command line
         */
  max_len = atoi ( argv[1] );

        /*
         * process text line-by-line
         */

  line_len = 0;
  while ( fgets( buffer, 200, stdin ) != NULL )
  {
    del_nl ( buffer );

    i = 0;
```

```
while ( buffer[i] != '\0' )
{
    /*
     * increment i until a non-space character is
     * found, or end-of-string
     */

  while ( isspace ( buffer[i] ) )
  {
    ++i;
  }

    /*
     * if not end-of-string, then
     * another word begins at i
     */

  if ( buffer[i] != '\0' )
  {
    word_len = 0;
    j = i;
    k = 0;

      /*
       * copy word to word buffer. Stops when j
       * is at a blank or end-of-string.
       */

    while ( buffer[j] != '\0' && ! isspace ( buffer[j] ) )
    {
      word_buffer[k] = buffer[j];
      ++j;
      ++k;
    }

      /*
       * k is the number of non-blanks transferred.
       */

    word_buffer[k] = '\0';

    word_len = k;
    if ( line_len == 0 )
      new_len = word_len;
    else
```

```
          new_len = line_len + word_len + 1;

        if ( new_len <= max_len )
        {
          if ( line_len == 0 )
            printf("%s", word_buffer);
          else
            printf(" %s", word_buffer);
          line_len = new_len;
          i = j;
        }
        else
        {
          printf( "\n%s", word_buffer );
          line_len = word_len;

          /*
           * The word printed was in buffer[i..j-1].
           * Therefore the next word, if any,
           * begins somewhere after j.
           */

          i = j;
        }
      }
    }                 /* search from i for another word ... */
  }                 /* read the next line */
  printf("\n"); /* add a final newline */
}
```

Sample run:

```
file tt:
We know that you highly esteem the kind of Learning taught in those
Colleges, and that the Maintenance of our young Men, while with
etcetera

%a.out 25 < tt
We know that you highly
esteem the kind of
Learning taught in those
Colleges, and that the
Maintenance of our young
Men, while with etcetera
%
```

26 Pointers and arrays

- C has **pointers.** A variable can contain the address of some piece of data: rather than containing data, it **'points'** to the data.
- To declare a pointer, use *:

  ```
  int * a, b, * c;
  ```

  defines `a` and `c` to be of type *pointer to* `int`, and `b` to be of type `int` as usual.
- Notice that each * applies *only to one* variable, even in a list of variables.
- When `x` is a pointer variable of type `double`, for example, then

  ```
  * x
  ```

  is the value of the double-precision number whose address is in `x`.
- C has a very odd convention:

  $\boxed{\text{arrays are pointers}}$

  and

  $\boxed{\text{array indexing can be used with pointers}}$
- In other words, if `a` is defined as an array, say `double a[15]`, **gcc** reserves a block of memory (15×8 bytes) to hold the array, and its *starting address* is stored in `a`.
- On the other hand, if `b` is declared as a pointer, say `double * b`, then `b` can be treated as an array, but it could be in a random piece of memory: no memory is reserved.
- Usually,

  ```
  char * a
  ```

  is used when `a` is a character string.

Example.

```
char * argv[]
```

declares `argv` to be an array of character strings.

```
argv[1][2]
```

would be the 3rd character (count begins at 0) of `argv[1]`.
Again, in

```
int a[10], *b, c[9]
```

the *values* of `a,b,c` will be addresses of integers. **But** storage will be reserved for `a,c`; while `b[14]` is accepted by C, it refers to some random piece of memory.

Also, C does not remember array bounds: `a[14]` is accepted, though it is outside the area reserved for `a`.

Summary.

- Use asterisks when declaring pointers

  ```
  int a[12], b, *c, d[15], *e, f, *g;
  ```

  ```
  a an integer array
  b an integer
  c pointer to integer
  d an integer array
  e pointer to integer
  f integer
  g pointer to integer
  ```

- In the above examples, $a, c, d, e,$ and g are all pointers Only a and d have memory reserved for them. You can assign an array or a pointer to a pointer,

  ```
  c = a;
  g = c;
  ```

  but you cannot assign anything to an array

  ```
  a = d;
    is impossible.
  ```

- Use an asterisk to get the value stored at an address:

  ```
  *e is the value stored at e
  *a is the value stored at a.  It is the same as a[0].
  ```

- Pointers are used to implement ‘call by reference.’ See below.

- `NULL` (declared in `stdio.h`) is the ‘null pointer value.’ For example, it is returned by `fgets()` when end-of-data has been reached.

- When an argument is a pointer, you pass it an address, such as in

  ```
  scanf("%d %d", &m, &n);
  ```

- Remember that under the rules of C, array arguments are effectively ‘call by reference.’

Using pointers for call-by-reference.

```
#include <stdio.h>
#include <ctype.h>
void count_alpha_numeric ( char * s, int * alpha, int * digits )
{
  int i;
  * alpha = * digits = 0;
  for (i=0; s[i] != '\0'; ++i)
  {
    if ( isdigit ( s[i] ) )
      ++ * digits;
    else if ( isalpha ( s[i] ) )
      * alpha = * alpha + 1;
  }
}

main()
{
  int a, d;
  char s[] = "0a1b2___cdea  ";
  count_alpha_numeric ( s, &a, &d );
  printf("%d alphabetic, %d digits\n", a, d);
}
```

27 Memory allocation, casts, and strings

There are 3 functions (use `stlib.h`) we can use for getting areas of storage:

- `malloc (), free ()` not covered in these lectures.
- `calloc ( int n, int s )` reserves $n \times s$ bytes of memory, **initialises** them to 0, and returns the address where they start.

There is a built-in function (sort of function) **sizeof (type)** which returns the number of bytes of storage occupied by an object of the given type.

To **cast** an expression is to convert it to another type. This is done with pointers to make types match. A **cast** is a type description in parentheses.

For example, `(double) 22` converts the integer expression to a double. More about this later.

Suppose you want an array of n doubles. Here is a function which does this.

```
double * array ( int n )
{
  double * a;
  a = ( double * ) calloc ( n, sizeof ( double ) );
  return a;
}
```

- `calloc( int n, int s )` obtains a block of $n * s$ bytes of free storage,
- **initialised** to all zeroes,
- and returns the address of the block.
- The type returned by the `calloc()` function is the most general pointer type possible: `void *`.
- Hence, to satisfy the rules of C, it is necessary to 'cast' it to the correct array type.

(27.1) Recall the 4 string-handling routines already mentioned

```
#include <string.h>
int strlen ( char * str )
void snprintf(<buffer>,<size>,<format>,<item>,...,<item>)
int strcmp (char * a, char * b )
int strncmp ( char * a, char * b, char * len)
        /* compares up to len characters */
```

Here is a program which copies strings to an array and prints them in reverse order.

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

void decr ( char * str )
{
  int len = strlen ( str );
  if ( len > 0 && str[ len-1 ] == '\n')
    str [ len-1 ] = '\0';
}
        /*
         * copy_string allocates storage for a copy
         * of the string, uses snprintf() to copy
         * the string to that storage, and returns
        * the copy.
        */
char * copy_string ( char * x )
{
  int len = strlen ( x );
  char * copy;

  copy = (char *) calloc (1, len+1 );
  snprintf ( copy, len+1, "%s", x );
  return copy;
}

main()
{
  char * line[1000];
  int count,i;
  char buffer[200];

  count = 0;
  while ( count < 1000 && fgets ( buffer, 200, stdin ) != NULL )
  {
    decr ( buffer );
    line[count] = copy_string ( buffer );
    ++count;
  }

        /* print in reverse order */

  for ( i=count-1; i>=0; --i )
    printf ("%s\n", line[i] );
}
```

27.1 2-dimensional arrays

We have created 1-dimensional arrays with no difficulty. Two-dimensional arrays are a lot more tricky.

```
  double a[10][10];
```

is a typical 2-dimensional array definition. How can we use pointers to produce arrays of flexible sizes? First of all the type must describe an array of arrays of doubles. Translating 'array' into 'pointer' we see that the appropriate type is `double * *`

```
  double ** c;
```

Can we create `c` like this?

```
  c = ( double * * ) calloc ( m, n * sizeof ( double ) );
```

No. Suppose that $m = n = 10$ and c is allocated 100 doubles beginning at address 40, say. Then

```
  c = 40
  c[0] = 0
  c[0][0] = ???
```

What we must do is create 10 separate arrays of size 10, and make `c` into an array of 10 pointers, giving the start of each 1-dimensional array.

Here is one way to do it.

```
  c = ( double * * ) calloc ( 10, sizeof ( double * ) );
  c[0] = ( double * ) calloc ( 10, sizeof ( double * ) );
  c[1] = ( double * ) calloc ( 10, sizeof ( double * ) );
etcetera

  Then c is an array of 'rows,' and each row is an array
of 10 doubles.  So for 0 <= i,j < 10,
  c[i][j]
    is the j-th entry in the i-th row of c.
```

These ideas lead to a matrix creation function

```
double * * mat ( int m, int n )
{
  int i;
  double * * mt;

  mt = ( double * * ) calloc ( m, sizeof ( double * ) );
  for (i=0; i<m; ++i)
  {
    mt[i] = ( double * ) calloc (n, sizeof ( double ) );
  }
}
```

In practical terms, memory allocation is a bit expensive in time and in space. Here is another version of the same function, more efficient because it uses 2 `callocs()`, not $m + 1$.

```
double * * mat ( int m, int n )
{
  int i;
  double * * mt;
  double * pool;

  mt = ( double * * ) calloc ( m, sizeof ( double * ) );
  pool = ( double * ) calloc ( m*n, sizeof ( double ) );
  for (i=0; i<m; ++i)
  {
    mt[i] = & ( pool [ i * n ] );
  }
}
```

28 Structures

28.1 Structures

It is possible to collect data into packages, called ‘structures.’ For example, the following declaration would be intended for complex numbers

```
  struct {double re, im;} z;
```

The variable z would be stored in 16 bytes, and its two components — which are double-precision numbers — would be referred to as

```
  z.re
  z.im
```

respectively. This ‘struct {etcetera}’ is a new kind of **type.** Usually, one introduces the type via **typedef**:

```
typedef struct { double re, im; }
  COMPLEX;

  COMPLEX x;
```

Note: the components of a `struct` are usually called *fields*.

28.2 Equality, assignment, routine arguments

A structure is a kind of generalised array, in that it has different elements, not too many, each with a different name. The C convention in which arrays are just pointers does not extend to structures.

```
void add ( COMPLEX a, COMPLEX b, COMPLEX * c )
{
...
}
...
  COMPLEX a,b;
  a.re = a.im = 0;
  b = a;
```

If ‘add’ is to add a and b and return the result in c, c *must* be a pointer. It is optional whether a and b are pointers: here they aren’t.

Best practice. There is no virtue in ‘call by value’ structure arguments; it is best to use pointer arguments.

Allocating structures. The `sizeof` pseudo-function can be used.

```
  COMPLEX * z;
  z = (COMPLEX *) calloc ( 1, sizeof ( COMPLEX ) );
```

Why should $*$ be needed then not needed? Think about it. Suppose that the `COMPLEX` structure has size 16 (which is almost certainly true). This says: allocate 16 bytes, and return the *address* — so the cast converts to ‘pointer to COMPLEX’ but **sizeof** returns the size of a COMPLEX object, 16, not the size of a pointer, which is 4 or 8.

28.3 Matrices

Interlude. Pointer notation. If `x` if of type `STR *` where `STR` is a structure, then to access field `n`, say, in the object addressed by `x`, the basic notation is

```
    (*x).n
```

but an alternative form is preferred:

```
    x->n
```

These forms are equivalent and interchangeable.

Here is a structure for matrices.

```
typedef { int height, width; double ** entry;}
MATRIX;
```

The first thing is to write a routine to create a matrix

```
MATRIX * zero_matrix ( int m, int n )
{
  MATRIX * mat = calloc ( 1, sizeof ( MATRIX ) );
  mat->height = m;
  mat->width = n;

  ... create the matrix entries as described earlier ...
  ... using calloc, all entries are initialised to 0 ...

  return mat;
}
```

The routine returns a *pointer*. In C, originally at least, structures could not be returned by functions.

Here is a routine to read a matrix from standard input. It creates and returns the matrix (pointer).

```
MATRIX * read_matrix ()
{
  int m, n, i, j;
  MATRIX * mat;

  scanf ( "%d %d", &m, &n );
  mat = zero_matrix ( m, n );
  for (i=0; i<m; ++i)
  for (j=0; j<n; ++j)
  {
    scanf ( "%lf", & ( mat[i][j]) );
  }

  return mat;
}
```

We need a routine to read an 'augmented matrix', storing the last column in a vector.

```
void read_aug_matrix ( MATRIX **a, VECTOR **b )
{
  int m, n, i, j;
  MATRIX * mat;
  VECTOR * vec;
  double x;

  scanf ( "%d %d", &m, &n );
  mat = zero_matrix ( m, n-1 );
  vec = zero_vecctor ( m );

  for (i=0; i<m; ++i)
  for (j=0; j<n; ++j)
  {
    scanf ( "%lf", & ( x ) );
    if ( j < n-1 )
      mat->entry[i][j] = x;
    else
      vec->entry[i] = x;
  }

  *a = mat;
  *b = vec;
}
```

29 The runtime stack and recursion

There are four kinds of variable: *local*, *routine argument*, *static*, and *global*.

The local variables and routine arguments are stored on the *runtime stack*. When a routine (or function) xxx begins, a *stack frame* is created to contain all the local data (including the routine arguments) for the routine.

Roughly speaking, this area of memory is called a ‘stack’ because it can grow and shrink.

For example

```
main begins:

  main calls a
  main ‘suspends operation’
      a begins with a new frame ‘pushed’ onto the stack.
         ...
      a calls b:
      a suspends operation
         b begins with a new stack frame.
                .....
         b ends
      a resumes....

      a ends
  main resumes.. etcetera
```

For example.

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

main()
{
  printf("gcd(276,42)=%d\n", gcd(276,42));
}
```

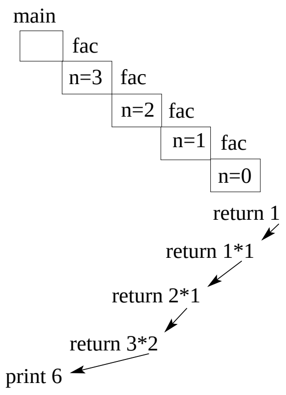
The ‘staggered’ layout below is to emphasise the rôle of the runtime stack; there are several different versions of m and n.

Our indenting policy is: the actions of a particular run of a routine are headed by ‘call ... ’ and terminated by ‘... returns’ or ‘... returns value.’ Between these lines they are indented a few columns.

```
main
  calls gcd(276,42)
  gcd
  | m 276 n 42
  | calls gcd(42,24)
  | gcd
  | | m 42 n 24
  | | calls gcd(24,18)
  | | gcd
  | | | m 24 n 18
  | | | calls gcd(18,6)
  | | | gcd
  | | | | m 18 n 6
  | | | | calls gcd(6,0)
  | | | | gcd
  | | | |   m 6 n 0
  | | | | gcd returns 6
  | | | gcd returns 6
  | | gcd returns 6
  | gcd returns 6
  gcd returns 6
  main prints:
gcd(276,42)=6.
```

Here is the ‘factorial’ example.

```
int fac( int n )
{
  if ( n == 0 )
    return 1;
  else
    return n * fac ( n-1 );
}
main()
{
  printf("3! = %d\n", factorial(3));
}
```



Here is a recursive procedure which behaves oddly — remember that static variables behave like global variables except they are private to the routine.

```
#include <stdio.h>
void printfac ( int n )
{
  static int x = 1;
  if ( n <= 1 )
    printf("%d\n", x);
  else
  {
    x *= n;
    printfac ( n-1 );
  }
}
main()
{
  printf("3! = ");
  printfac(3);
}
```

We shall keep x on the right to emphasise that it is not in the stack frame.

```
main prints
3! =
main calls printfac
  printfac (3)
                                x = 3*1 = 3
    printfac calls
      printfac (2)
                                x = 2*3 = 6
         printfac calls
         printfac (1)
            which prints x, i.e., 6, completing
            the line:
3! = 6
         printfac returns
      printfac returns
    printfac returns
  printfac returns
main ends
```

30 Conversions, casts, and pointers

C performs automatic *type conversions*, when an expression contains subexpressions of different types.

- Strangely, `char` expressions are considered a kind of integer, and where necessary they are 'promoted' to `int` expressions. **Careful.** They may or may not use sign extension, so the promoted value can be negative.

  This happens on Intel chips — on the maths machines.

  It is not usually noticeable, because The usual ASCII characters are between 0 and 127, and the high-order bit is zero.

- You can get around it using the `unsigned` qualifier. There is a data-type

  `unsigned char`

  which does not cause sign extension.

- Shorts are always promoted to ints.
- Floats are always promoted to doubles.
- Ints are promoted if necessary to longs — this makes no difference on the maths machines.
- When ints and doubles are mixed, the ints are promoted to doubles.

When assigning doubles to ints, etcetera

- Assigning double to float: the value is computed and rounded.
- Int to short and int or short to char: the high-order bits are dropped.
- Doubles and floats to int: values are rounded toward zero. That is, 2.5 rounds to 2, and -2.5 rounds to -2.

Type casts are a way to force conversions. The notation is

```
(<casting type>)  <expression>
```

For example, if x is a `double`, then `(int) x` is x rounded up or down to integer depending on sign. Not sure what happens when x is out of range.

`(double) 2` and `2.0` are the same.

(30.1) Pointers. Casts convert expressions to a plausibly equivalent value in a specified type.

There is another place where they are very important. Addresses are 4 or 8 bytes long depending on the machine. **Memory allocation** functions reserve pieces of memory and return the value in a fixed type, which used to be `int,` but now is a more cautions `void *` — meaning pointer to object of unspecified type.

To satisfy gcc, it is necessary to 'cast' this to the required type. For example,

```
char * a;

a = ( char * ) malloc ( 121 );
```

31 Files

Input/output redirection. You can arrange that `scanf()` and `fgets (..., stdin)` read from a file rather than a terminal, and make `printf()` write to a file:

```
%a.out < my_input
%a.out > my_output
%a.out < my_input > my_output
```

FILEs can be declared in C (stdio.h contains the definitions, I think).

```
FILE * myfile;
```

There are the three ‘standard’ files **stdin, stdout, stderr.**

```
  printf ( .... ); and
  fprintf ( stdout, .... );
are the same, and
  scanf ( .... ); and
  fscanf ( stdin, .... )
are the same.
```

Apart from these, a file must be **opened** before it can be read from or written to, as follows

```
file = fopen ( <file name>, "r" ); /* for reading */
file = fopen ( <file name>, "w" ); /* for writing */
file = fopen ( <file name>, "a" ); /* for appending */
```

`fopen()` **returns a memory address** where details about the file are stored; NULL if it was impossible to open the file. A file should be **closed** after use:

```
fclose ( file );
```

If a file was opened for reading, it is unnecessary to close it, but does not harm.
If a file is opened for writing/appending and not closed,

the updates will be lost.

You can read from a file using `fscanf()` and `fgets()`.

Note about fscanf(). It returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching fail- ure. Zero indicates that, while there was input available, no conver- sions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a ‘returned if an input failure occurs before any conversion such as an end- of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned. (**EOF** is defined in `stdio.h`, its value is -1, I think.)

```
fgets ( <buffer>, <buffer length>, <file> );
```

As already mentioned, `fgets()` returns the address of the buffer, if data was read, and returns `NULL` if end-of-data was reached.

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

        /* program reads lines and
         * prints them with an extra space.

void decr ( char str[] )
{
  int i;

  for ( i=0; str[i] != '\0'; i = i+1 )
    if ( str[i] == '\n' )
      str[i] = '\0';
}

main ( int argc, char * argv[] )
{
  char buffer[100];
  FILE * infile;
  FILE * outfile;


  infile = fopen ( argv[1], "r" );
  outfile = fopen ( argv[2], "w" );


  while ( fgets ( buffer, 100, infile ) != NULL )
  {
    decr ( buffer );
    fprintf (outfile, "%s\n\n", buffer );
  }

  fclose ( infile );
  fclose ( outfile );

}
```

When reading from a named file, you can *read the file more than once* using the **rewind()** routine. For example, it might be necessary to count the data items and rewind the file in order to process them again:

```
        /* Take the average of a list of numbers, but
         * first counting them.
         */
```

```
#include <stdio.h>

main (int argc, char * argv[] )
{
  int n;
  FILE * file;
  double x, sum, average;

  file = fopen ( argv[1], "r" );
  sum = 0;
  n = 0;
  while ( fscanf( file, "%lf", & x ) == 1 )
  {
    sum += x;
    ++ n;
  }

  average = sum/n;

  printf("%d numbers read, average is %f\n",  n, average);

  rewind ( file );

        /*
         * Now you can scan the numbers
         * again --- but this time you
         * know the average.
         */
}
```

32 Multi-file compilation

You can compile a C program from several files

```
gcc ninth.c matrix9.c
```

The routines in `matrix9.c` are

```
static MATRIX * zero_matrix ( int m, int n );
static VECTOR * zero_vector ( int n );
void read_aug_matrix ( FILE * file, MATRIX **a, VECTOR **b );
void print( FILE * file, MATRIX * a, VECTOR * b );
```

```
static void subtract ( MATRIX * a, VECTOR * b, int i, double s, int j );
static void scale ( MATRIX *a, VECTOR *b, int i, double s );
static void swap ( MATRIX * a, VECTOR * b, int i, int j );
void reduce ( MATRIX * a, VECTOR * b, MATRIX ** uu, VECTOR ** yy );
VECTOR * back_substitute ( MATRIX * u, VECTOR * y );
void print_vector( FILE * file, VECTOR * x );
```

The **static** keyword means *private to matrix9.c.* These routines are not meant to be 'exported.' In order to use the matrix routines in `ninth.c`, the following **header file** should be used (it is on the web for the ninth assignment).

```
matrix9.h:

typedef struct { int height; double * entry; }
  VECTOR;

typedef struct { int height, width ; double ** entry; }
  MATRIX;

void read_aug_matrix ( FILE * file, MATRIX **a, VECTOR **b );
void print( FILE * file, MATRIX * a, VECTOR * b );
void reduce ( MATRIX * a, VECTOR * b, MATRIX ** uu, VECTOR ** yy );
VECTOR * back_substitute ( MATRIX * u, VECTOR * y );
void print_vector( FILE * file, VECTOR * x );
```

Here is a template for `ninth.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrix9.h"

main( int argc, char * argv[] )
{
  MATRIX *a, *u;
  VECTOR *b, *y, *x;
  FILE * in, * out;

        /*
         * Open in, out
         */

  read_aug_matrix ( in, &a, &b );
```

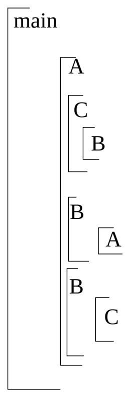


Figure 2: The life of variables

```
  fprintf ( out, "Input matrix\n");
  print( out, a, b );

  reduce ( a, b, &u, &y );

  fprintf (out,"Reduced matrix\n");
  print( out, u, y);
  x = back_substitute( u, y );
  fprintf (out, "Solution\n");
  print_vector (out,x);

        /*
         * etcetera
         */
}
```

33 Scope and lifetime of variables

Variables and routines have scope and variables also have lifetime.

- **Scope** within a file or a routine, below the point where the routine/variable is introduced.
- **Lifetime** is a single run of the routine for local non/static variables and routine arguments. It is the duration of the entire program for global and static variables.

	Scope	Lifetime	Initialised how often?
local	private to routine	lasts for a single run of the routine	initialised (if at all) at start of every run of the routine
global	accessible from all routines	lasts for entire run of the program	initialised once
routine argument	private to routine	single run	start of every run
static local variable	private to routine	lasts entire program	intialised once

Prototypes allow routines or variables to be introduced without a full definition. Declaring variables with the keyword **extern** is a form of prototyping.

Keyword **static** applies also to global variables and routines. For them, it means **private to the file,** as opposed to external scope.

Variables introduced with **extern** are prototypes and need to be defined in the same file or some other file.

```
int a[] = {1,2,3};
extern int c;
                        /*
                         * c must be defined in this file or
                          * another with multifile compilation
                         */

static int d;                /* private to the file */
void e ( int n )      /* prototype */
{
  int b = a[1];                /* OK, within scope of a; */
}
static void f (){}    /* private to the file */

int c = 1;                /* ok but odd */

int main(){};                /* correct but pointless */
```

34 Operator precedence

We know the BODMAS rules for arithmetic operators. C is full of operators, and they have carefully defined ‘precedence.’

- The highest precedence operators are evaluated left-to-right. Otherwise they have equal precedence. In this and other groups, where ‘right to left’ or ‘left to right’ is stated, this fixes the precedence where otherwise they have equal precedence.

- [] (i.e., accessing array element)
- . (structure member)
- $->$ (structure member through pointer)
- Postfix increment/decrement

- Next, right to left:
  - Prefix increment/decrement
  - Casts
  - $*p$ (the object stored at location p)
  - $\&$ address
  - `sizeof()`
- $+/\%$ multiplication, division, remainder modulo
  Left to right.
- $+-$ addition, subtraction
  Left to right.
- $<, <=, >=, >$ relations
  Left to right.
- $==, \; != $ relations
  Left to right.
- $\&\&$ logical AND
  Left to right.
- $||$ logical OR
  Left to right.
- $=, \; + =, \; - =,$ etcetera Assignment and assignment operators
  Right to left.

Examples.
Disambiguate the following expressions by inserting parentheses, and say whether the expression is meaningful (legal), assuming the variables have suitable types.

```
  (i) while (   *x++!='\0'   )..
 (ii) a = b = c == 0
(iii) a = b == c = 0
 (iv) a = b = c == d && e || f || g
  (v)  * x[3] -> y[4]
  (i) while (   *x++!='\0'   )..
       while (  ( *(x++) ) != '\0' )..
legal
 (ii) a = b = c == 0
        a = ( b = ( c == 0 ))
legal
(iii) a = b == c = 0
        a = ( ( b == c ) = 0 )
illegal
 (iv) a = b = c == d && e || f || g
      a = ( b = ((( ( c == d ) && e  )|| f )|| g ) )
legal
  (v)  * x[3] -> y[4]
        * ( (x[3]) -> (y[4]) )
illegal
```

35 May 2014 syllabus

35.1 Marks breakdown

- Questions will be based on the topics given below, on the programming assignments, and the quizzes.

35.2 Topics

- Data types: char, short, int, long, float, double, address
- 2s complement short and int numbers, addition, and subtraction.
- Programming elements: for-loops, assignments, conditions, while, if-then-else.
- Command-line arguments, atoi, atof.
- Files: fopen, fclose.
- Printf, scanf, fgets, fscanf, fprintf. Difference between printf and scanf.
- Array and string initialisation.
- Functions and subroutines: writing fairly simple functions and subroutines.

  Simulating given functions and subroutines, which might be recursive.
- Single and double precision floating point numbers. (You should know the $1 : 8 : 23$ format for single precision. Double precision conversions will not be asked.)

- Arrays, 1- and 2-dimensional. Calculating the size of an array and the addresses of array elements, for which you should know the length of char (1), short (2), int (4), float (4), double (8), address (4). You must know these lengths: they would not be given in the exam.
- Static and automatic variables in routines: automatic variables are stored on the runtime stack.
- Global variables, keywords
  - `static` affects both *lifetime* (throughout the program run) and *privacy*: static local variables are completely private, static global variables and routines are private to the file in which they occur.
  - `extern` before a declaration is when the variable type is required but the variable is defined elsewhere, probably in another file. `extern` declarations do not reserve space. These are summarised at the end of Section 32.
- Arithmetic expressions
  - Order of evaluation (precedence), only up to BODMAS rule and the fact that relations have lower precedence. For example

    ```
    2 == 3+4
    ```

    is evaluated as follows: $3 + 4$ first, having higher precedence than `==`, so the expression becomes `2 == 7`; this evaluates to 0 (meaning false).
  - Conversion, as for example where

    $$1 + 1.0/2$$

    evaluates to 1.5, hecause $1.0/2$ is evaluated as double, and 1 is converted to double before adding.

    $$1 + ((\text{double})1)/2;$$

    has the same effect.
  - Casts, such as `(int) 3.4` and `(char *) malloc ( n+1 )`.
- Malloc and calloc and strings, e.g., for copying strings.
- Structures applied to matrices, malloc and calloc.
- Use of header files and multi-file compilation, as in the last programming assignment.