POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Problem-solving with scalar data

FROM PROBLEM TO PROGRAM: INTRODUCTION TO PROBLEM-SOLVING IN C LANGUAGE

Problems: overview

- numeric
- encoding/decoding
- text-processing
- verification, filtering and sorting

Without using arrays (only scalar data)

With conditional constructs (simpler problems) and/or iterative constructs.

Scalar data

- They include
  - numbers
  - Characters/strings (as a whole)
  - Heterogeneous aggregates (`struct`)
- Arrays (vectors and matrices) are excluded
- We include:
  - strings, used as a whole entity (word/sentence), and not as arrays of characters
  - `struct`, in that they are not a numerable aggregate of elements

Types of problems we handle in this Section

- In case the solution is iterative (i.e., it involves a loop), it is possible to iterate on the current i-th datum **without** remembering all the previous ones
- We may need to remember more than one single datum (for example, the one immediatly before the current one), but not ALL the previous data at once. → it is not necessary to store data in arrays

Numeric problems

- Problems of algebra, geometry, statistics, etc., typically characterized by:
  - Numeric data (real or integer)
  - Mathematical expressions (formulas) or sequences/iterations of mathematical operations
- Advantage: easy to convert into a program
  - Mathematical problems are typically already formulated in a rigorous and structured form –> it is easy to pass from from the problem statement to the algorithm

Caveat

- when translating the mathematical formulation to a C program, you need to take into account:
  - Finite representation of numbers (overflow/underflow problems)
  - Rounding/truncation problems and integer-real number conversions

Non-iterative numeric problems

- Characterized by selecting a different subproblem (and, so, a different mathematical operation) based on a set of conditions
- Typically, the different subproblems are selected by using **if construct** (or nested if, whenever necessary). It is rare that we use *switch* construct, because the selector needs to be an integer variable in this case, which limits the usability a lot
- Even though different configurations of the if construct are possible, the solution (data structure & algorithm) is simple and straightforward
- May be possible having to deal with type conversions, casts, rounding/truncation problems

II degree equation

- Formulation:

  Given three coefficients (a, b, c) of the equation

  $$ax^2 + bx + c = 0$$

  Determine the roots (solutions) of the equation, distinguishing cases of impossible equation, indetermined equation, first degree equation, second degree equation with two distinct real solutions, two coincident solutions, complex conjugate solutions

- Solution: typical selection problem with conditional constructs, where each case corresponds to a different mathematical operation and/or output message

II degree equation

- Data structures: scalar variables, float (or double, if deemed necessary)
  - coefficients: a, b, c
  - determinant: delta
  - solutions: x0, x1 for real solutions, re, im for complex ones
- Algorithm: select between 5 different cases by means of if … else constructs (either simple or nested)
- Caveat: The selection is based on the values of the determinant, and not on a simple integer selector → switch construct does not apply

Solution version 1: non-nested `if`

```
#include <math.h>

#include <stdio.h>

int main(void) {

  float a,b,c,delta,x0,x1,re,im;

  printf("Insert coefficients (a b c): ");

  scanf("%f%f%f",&a,&b,&c);

  if (a==0 && b==0 && c==0)

    printf("Indetermined equation\n");

  if (a==0 && b==0 && c!=0)

    printf("Impossible equation\n");

  if (a==0 && b!= 0) {

    printf("I degree equation\n »);

    printf("Solution: %f\n", -c/b);

  }
```

```
  if (a!=0) {
    delta = b*b-4*a*c;
    if (delta==0) {
      x0 = (-b)/(2*a);
      x1 = (-b)/(2*a);
      printf("2 real coincident solutions: ");

      printf("%f %f\n",x0,x1);
    }
    if (delta > 0) {
      x0 = (-b-sqrt(delta))/(2*a);
      x1 = (-b+sqrt(delta))/(2*a);
      printf("2 real distinct solutions: ");

      printf("%f %f\n",x0,x1);
    }
    if (delta < 0){
      re = -b/(2*a);
      im = sqrt(-delta)/(2*a);
      printf("2 compl. Conjugate sol.: ");
      printf("x0=%f-i*%f ",re, im);

      printf("x1=%f+i*%f\n", re, im);
    }
  }
}
```

Solution version 2: nested `if`

```c
#include <math.h>

#include <stdio.h>

int main(void) {

  float a,b,c,delta,x0,x1,re,im;

  printf("Insert coefficients (a b c): ");

  scanf("%f%f%f",&a,&b,&c);

  if (a==0) {

    if (b==0) {

      if (c==0)

        printf("Indetermined equation\n");

      else

        printf("Impossible equation\n");

    }
```

```c
     else {

      printf("I degree equation\n »);

      printf("Solution: %f\n", -c/b);    }

  }

  else {

    delta = b*b-4*a*c;

    if (delta == 0) {

      x0 = (-b)/(2*a);

      x1 = (-b)/(2*a);

     printf("2 real coincident solutions: ");

     printf("%f %f\n",x0,x1);

    }

    ...

  }
```

Solution version 2: nested `if`

```
#include <math.h>
#include <stdio.h>
int main(void) {
  float a,b,c,delta,x0,x1,re,im;
  printf("Insert coefficients (a b c): ");
  scanf("%f%f%f",&a,&b,&c);
  if (a==0) {
    ...
  }
  else {
    delta = b*b-4*a*c;
    if (delta == 0) {
    }
```

```
    else /* if delta != 0 */
      if (delta > 0) {
        x0 = (-b-sqrt(delta))/(2*a);
        x1 = (-b+sqrt(delta))/(2*a);
        printf("2 real distinct solutions: ");
      }
      else { /* delta < 0 */
        re = -b/(2*a);
        im = sqrt(-delta)/(2*a);
        printf("compl.conj.: \n
                x0=%f-i*%f x1=%f+i*%f\n",
                re, im, re, im);
      }
  }
}
```

Area of a right triangle

- Formulation:
  - Right triangle → lengths of sides are a Pythagorean triple
  - Given length of sides (integers: a, b, c) of a right triangle, determine which one is the hypotenuse
  - Compute the area of the triangle
- Solution:
  - Hypotenuse is the longest side
  - Compute area (real number)

Area of a right triangle

- Data structure: scalar variables representing:
  - sides: a, b, c (type int)
  - area: area (type float)
- Algorithm: we need to select between 3 different cases (a, b or c maximum) to decide on the hypotenuse. Options:
  - Option 1 - Permutation of the sides, so that we always end up into the same configuration of a,b,c and we do not need to change mathematical expression
  - Option 2 - Select between 3 different area computations expressions (proposed solution).
- Caveat: in computing the area we need to perform a real number division (not an integer one!)

Solution version 1: 3 `if`, 6 comparisons

```c
#include <math.h>
#include <stdio.h>
int main(void) {
  int a,b,c;
  float area;

  printf("Sides of triangle (a b c): ");
  scanf("%d%d%d",&a,&b,&c);

  if (a>b && a>c) {
    printf("Hypotenuse is a\n");
    area = b*c/2.0;
  }
 else if (b>a && b>c) {
    printf("Hypotenuse is b\n");
    area = a*c/2.0;
  }
  else if (c>a && c>b) {
    printf("Hypotenuse is c\n");
    area = a*b/2.0;
  }
  printf("Area is %f\n", area);
}
```

Solution version 1: 3 `if`, 6 comparisons

```c
#include <math.h>
#include <stdio.h>
int main(void) {
  int a,b,c;
  float area;

  printf("Sides of triangle (
  scanf("%d%d%d",&a,&b,&c);

  if (a>b && a>c) {
    printf("Hypotenuse is a\n");
    area = b*c/2.0;
  }
```

```c
 else if (b>a && b>c) {
    printf("Hypotenuse is b\n");

  }
  printf("Area is %f\n", area);
}
```

Using a float constant (2.0) makes sure that we perform a division between float numbers, and hence that we obtain a float result.
We could have also done as follows:

area = (float)(b*c)/2.0;

or

area = (float)b*(float)c/2.0;

Solution version 2: 3 `if`, 3 comparisons

```
...

int main(void) {

  ...

  if (a>b) /* Hypotenuse is NOT b*/
    if (a>c) {
      printf("Hypotenuse is a\n"); area = ((float) (b * c)) / 2.0;
    }
    else {
      printf("Hypotenuse is c\n"); area = ((float) (a * b)) / 2.0;
    }

  } else /* Hypotenuse is NOT a */
    if (b>c) {
      printf("Hypotenuse is b\n"); area = ((float) (a * c)) / 2.0;
    }
    else {
      printf("Hypotenuse is c\n");area = ((float) (a * b)) / 2.0;
    }

  }

  ...

}
```

Iterative numeric problems

- Same as the previous category, with addition of iterations or repeated mathematical operations
- Examples:
  - Numerical successions or series (ex., Fibonacci numbers)
  - Geometric problems with polygons, sequences of points and/or segments
  - Iterative formulation of mathematical problems (ex, factorial computation)
  - maximum/minimum computation, summations, mean values, statistics on data sequences

N-th reduced of harmonic series

- Formulation:
  - Harmonic series: sequence of the natural numbers' reciprocals (1, 1/2, 1/3, …)
  - n-th reduced of the series is defined as follows

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

  - Write a C program that:
    - reads n from keyboard (multiple times)
    - if $n \leq 0$ terminates execution. Otherwise, it determines and prints the reduced Hn

- Solution:
  - Iteratively generate the terms of the sequence: 1, 1/2, 1/3, ...
  - Incrementally sum-up the terms of the sequence

N-th reduced of harmonic series

- Data structures: scalar variables to represent:
  - n parameter and an index i, to count iterations (i.e., terms of the successions) → type int
  - Sum HN, iteratively computed → type float
- Algorithm:
  - External loop to repeatedly acquire n (terminates execution if n<=0)
  - Per each valid n, internal loop to generate the succession of terms 1..n, and compute sum HN.

Solution

```
#include <math.h>
#include <stdio.h>
int main(void) {
  int n, i;
  float HN;
  printf("Number of terms (<=0 to END): ");
  scanf("%d",&n);
```

```
  while ( n>0 ) {
    /* compute and print HN */
    HN = 0.0;
    for (i=1; i<=n ; i++)
      HN = HN + 1.0/((float)i);
    printf("Result: %f\n", HN);

    printf("Number of terms (<=0 to END): ");
    scanf("%d",&n);
  }
}
}
```

Encoding/decoding problems

- Search problems where we need to recognize or generate numeric or non-numeric encodings
- Numeric encoding: can handle either integer or real numbers, base 2 or different bases. Typical problems:
  - Converting from one base to another (including reading/printing numbers in a certain base)
  - Operations on specific bases
- Non numeric encodings (for example, characters): can handle binary encoding of characters. Typical problems:
  - decoding/recognize internal codes
  - Change of encoding (re-encoding/cryptography): we need to know encoding rules and/or encoding tables.

Problems with numeric values

- Numbers in C are represented:
  - internally in base 2 (2’s complement, FP IEEE-754, ...)
  - externally (input/output) in decimal, octal, exadecimal
- Arithmetic operations are automatically performed in base 2. The only problems we need to take into account are overflow/underflow situations.

Problems with numeric values

- Numbers in C are represented:
  - internally in base 2 (2’s complement, FP IEEE-754, ...)
  - externally (input/output) in decimal, octal, exadecimal
- Arithmetic operations are automatically performed in base 2. The only problems we need to take into account are overflow/underflow situations.

The management of representation and numeric encoding is automatically done by the computer. Why do we need to Deal with encoding algorithms?

Problems with numeric values

- Handling explicitly the representation/encoding of numbers can be useful in a number of cases:
  - To use non-standard encodings (for example, base 4, 5 or others)
  - To modify the representation limits
  - To explicitly “decode” numeric representations
- Typical problems
  - Base conversions: one of the encoding can be the internal one (it can be even used as the intermediate passage between two other bases)
  - Explicit computation of arithmetic operations, working on individual digits

Problems with numeric values

- Encoding problems are often solved iteratively. We will start by handling algorithms that do not require the use of arrays
- Depending on the format, it may be needed to handle numeric digits, signs and/or exponents (given the base)
- It is possible to eventually perform arithmetic operations, dealing with representation of the individual digits

Pure binary encoding of an integer positive number

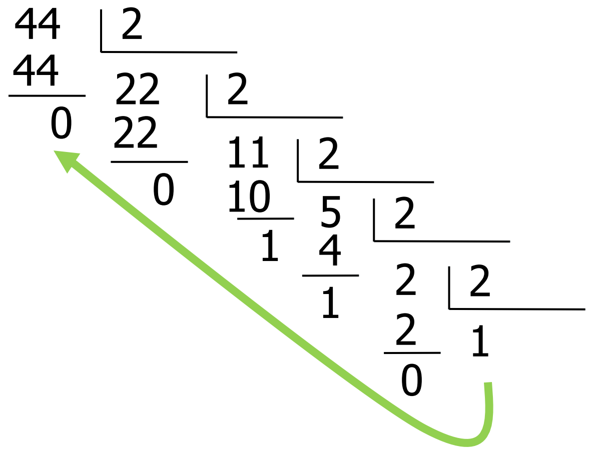
- Formulation:
  - Given an integer number (>=0), write a C program that determines its binary encoding and prints the individual bits of the binary conversion
- Solution:
  - The classic algorithm to generate a binary encoding is based on subsequent divisions by 2. Unfortunately, this algorithm generates the bits starting from the LSB up-to the MSB (i.e., the binary number is generated from right to left) → we would need an array to store the values
  - The generation of the bits starting from MSB can be done by iteratively computing the highest power of 2 that is <= of the number to convert

Pure binary encoding of an integer positive number

- Data structure: 2 integer variables:
  - Formal parameter (integer): n
  - A variable to store at each iteration a decreasing power of 2: p
- Algorithm:
  - First, we generate and store in p the largest power of 2 that is <= n
    - Start with p = 1
    - At each iteration, p = 2*p
    - Stop iterating when p>n. The last correct value of p is the one of the previous iteration.
  - Then, we generate the bits. At each iteration:
    - if n >= p $\rightarrow$ bit = 1, n = n-p
    - if n<p $\rightarrow$ bit = 0
    - p = p/2

Example: convert 44_{10} to base 2

Subsequent divisions:
We need an array



$44_{10} = 101100_2$

Algorithm from MSB to LSB:
We do not need an array

32: max power of 2 <= 44

32 <= 44	44 – 32 = 12 32/2 = 16	1
16 > 12	16/2 = 8	0
8 <= 12	12 – 8 = 4; 8/2 = 4	1
4 <= 4	4 – 4 = 0 4/2 = 2	1
2 > 0	2/2 = 1	0
1 > 0	1/2 = 0	0

Solution

```
void binary (int n) {
  int p;
  for (p=1; 2*p<=n; p=p*2);
  while (p>0) {
    if (p<=n) {
      printf("1");
     n=n-p;
    }
    else printf("0");
    p = p/2;
  }
  printf("\n");
}
```

Solution

```
void binary (int n) {
  int p;
  for (p=1; 2*p<=n; p=p*2);
  while (p>0) {
    if (p<=n) {
      printf("1");
     n=n-p;
    }
    else printf("0");
    p = p/2;
  }
  printf("\n");
}
```

Find largest power <= n

Solution

```
void binary (int n) {
  int p;
  for (p=1; 2*p<=n; p=p*2);
  while (p>0) {
    if (p<=n) {
      printf("1");
     n=n-p;
    }
    else printf("0");
    p = p/2;
  }
  printf("\n");
}
```

Per each p = decreasing power of 2
- If p is smaller than n, print 1 and subtract p from n

Solution

```
void binary (int n) {
  int p;
  for (p=1; 2*p<=n; p=p*2);
  while (p>0) {
    if (p<=n) {
      printf("1");
     n=n-p;
    }
    else printf("0");
    p = p/2;
  }
  printf("\n");
}
```

Per each p = decreasing power of 2
- If p is smaller than n, print 1 and subtract p from n

- otherwise, print 0

Complete solution: main + function binary

```
#include <stdio.h>
void binary(int n);   // prototype
int main(void) {
    int v;
    printf("Insert positive number: ");
    scanf("%d",&v);
    printf("Binary conversion: ");
    binary(v);   // function call
    return 0;
}
void binary(int n)    // function definition
{
    // see previous slide …
}
```

Conversion between bases

- Formulation:
  - Acquire from keyboard two integers b0 e b1 (between 2 and 9), corresponding to the initial and final value of the base
  - Iteratively acquire integer numbers (unsigned) in base b0 (maximum valid digit is b0-1, spaces and/or newlines are separators for the acquisition)
  - Each integer number needs to converted from base b0 to b1 and printed on video
  - The program terminates when the user inserts a non-valid number
- Solution:
  - Acquire b0 and b1 from keyboard
  - Iteratively acquire individual digits of the number to be converted from b0 to b1, in the form of characters
  - Convert to b1 (similar algorithm as for binary conversion), using conversion to decimal as intermediate step (b0-->b10-->b1)

Conversion between bases

- Data structures: integer variables
  - Two int variables for bases b0 and b1
  - A variable for the number (n), directly converted from base b0 to base 10, digit by digit
  - A variable p for decreasing powers of base b1 (same as in binary conversion, see previous example)
  - A variable end, used as a flag in the loop of acquisition of sequences

Conversion between bases

- Algorithm:
  - Acquire b0 and b1 from keyboard and initialize n=0
  - Iteratively acquire characters (one at a time). Per each character:
    - If it is a space or a newline, convert n to base b1, calling the function conversion
    - If it is a valid digit in base b0, convert character to a number (using ASCII rules) and update n (i.e., multiply n by b0 and add the acquired number)
    - If it is a non-valid number, terminate the program
  - The function conversion is similar to the binary function implemented before.

Solution (part 1)

```c
#include <stdio.h>
void conversion(int n, int b);

int main(void) {
  int b0, b1, n, p, digit, end=0;
  char c;

  printf("b0 (2..9): "); scanf("%d",&b0);
  printf("b1 (2..9): "); scanf("%d\n",&b1);

  n = 0;
```

```c
  while (!end) {
    scanf("%c",&c);
    if (c== ' ' || c== '\n') {
      conversion(n,b1); n=0;
    }
    else {
      digit = c - '0'; // converts char to number
      if (digit>=0 && digit<b0)
        n = b0*n + digit;
      else
        end=1;
    }
  }
}
```

Solution (part 2)

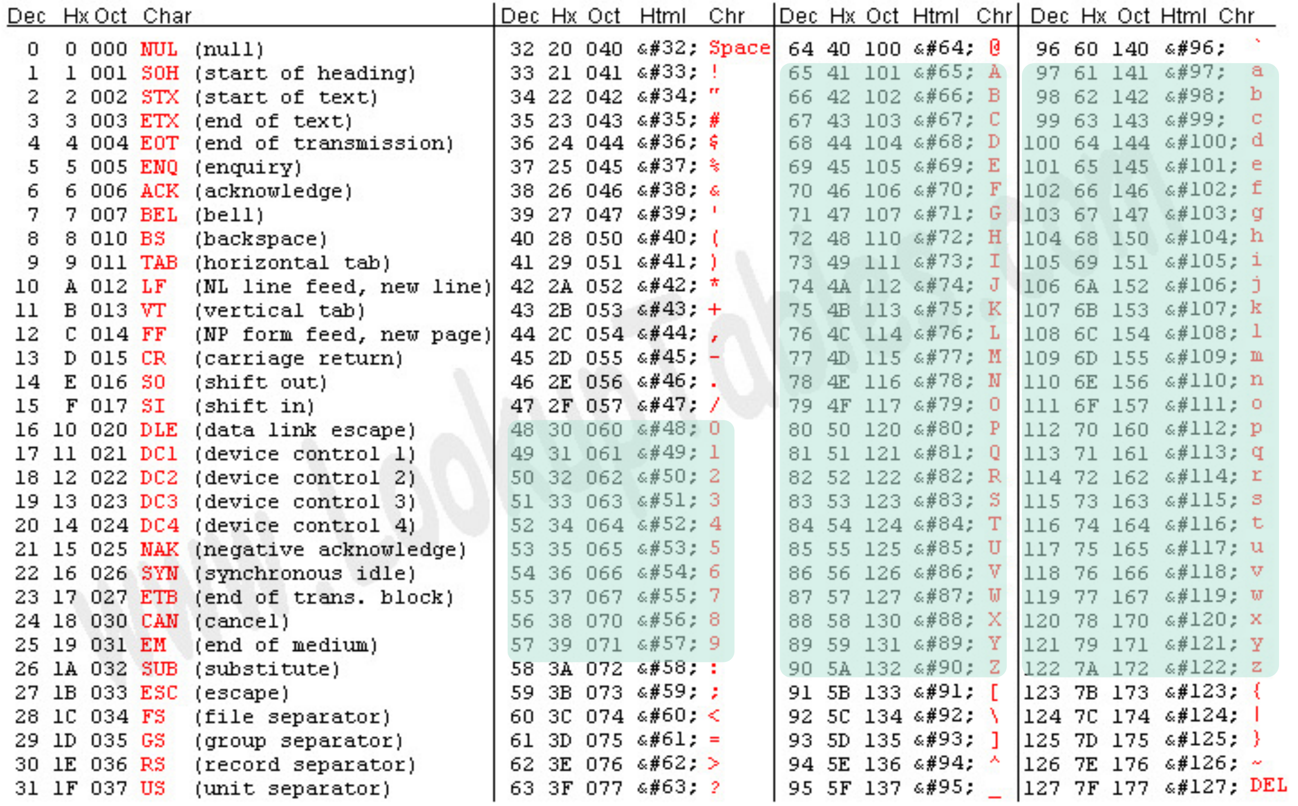
```
void conversion(int n, int b) {
  int p;

  for (p=1; b*p<=n; p=p*b);  // same as binary, but with b instead of 2
  while (p>0) {
    if (p<=n) {
      printf("%d",n/p); n=n%p;
    }
    else printf("0");
    p = p/b;
  }
  printf("\n");
}
```

Problems on non-numeric values

- In C characters are encoded based on ASCII standard:
  - internally ASCII uses 8 bit for representation
  - It is NOT necessary to know the codes. It is sufficient to remember few properties:
    - subsets of alphabetic (uppercase and lowercase) and numeric characters are contiguous (‘0’, ‘1’ = ‘0’+1, ‘2’ = ‘1’+1, etc.)
    - 'Z' and 'a' are not contiguous
  - externally (input/output), characters are visualized in a textual form

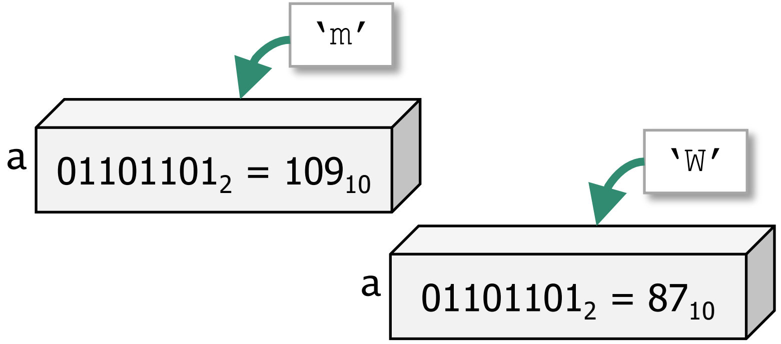
ASCII table



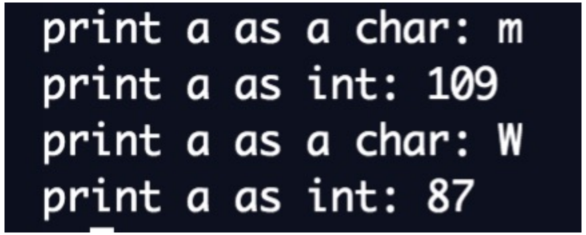
Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Tiype char

- Basic type for characters
  - Size = 8 bit = 1 Byte
  - As defined by ASCII



```
#include <stdio.h>
int main(void) {
  char a;
  a = 'm';
  printf(”print a as a char: %c\n", a);
  printf(”print a as int: %d\n", a);
  a = 'W’;
  printf(”print a as a char: %c\n", a);
  printf(”print a as int: %d\n", a);
return 0;
}
```



```
print a as a char: m
print a as int: 109
print a as a char: W
print a as int: 87
```

Operations on characters

- Arithmetic operations, comparisons of characters, ordering are done by using ASCII encoding.
  - For all the arithmetic operators, char operands are treated like 8bit integers (ASCII codes of the corresponding characters)
  - Examples:
    - 'C' - 'A' is 2
    - 'Z' - 'A' is 25
    - '5' – '0' is 5
- To convert a numeric character (for example, character '5’) to the corresponding numeric value, you need to subtract the ASCII code of ‘0’:

```
int digit; char c = '5';
digit = c – '0';
```

- To convert an alphabetic character from upper to lowercase or viceversa:

```
char c1 = ‘H’, c2 = ‘r’;
c1 = c1 – ‘A’ + ‘a’;   // c1 now contains ‘h’
c2 = c2 – ‘a’ + ‘A’;   // c2 now contains ‘R’
```

- It is possible to use relational operators (==, !=, >, <, >=, <=) : based on ASCII code, ‘A’ < ‘C’

Problems with character encoding

- Why?
  - It could be necessary to determine the ASCII code of a character:
    - You just need to implement a conversion of number to binary
  - You may want to implement a re-encoding to:
    - encrypt/deencrypt or compact/uncompact a text
- What kind of algorithms?
  - Encoding problems typically involve:
    - Iterations to handle sequences of characters
    - Handling individual characters by special manipulations

Simple encryption: example of problem

- Formulation:
  - Encrypt the content of a text file, storing the result in a second file
  - Encryption consists in modifying the encoding of the alphanumeric characters, following these rules:
    - Each numeric character n (‘0’..’9’ digit) needs to be converted to its 9’s complement (‘0’+‘9’-n):
    - Each alphabetic character ch is transformed by converting uppercase to lowercase and vice versa, and computing z’s complement ('a' + 'z' -ch)
- Solution:
  - Iteratively read characters one by one from input file
  - Based on type of character, apply suitable conversion
  - Print obtained character on output file

Simple encryption

- Data structure:
  - Two FILE pointer variables fpin, fpout respectively for input and output files
  - String namefile for file names
  - Char variable for reading and encyption of single characters
- Algorithm:
  - Acquire file names and open corresponding files
  - Iterative reading of single character from input file, conversion, print to output file
    - Conversion is made based on the category the character belongs to (numeric, alphabetic) and corresponding conversion rule. For example, the new code for a ‘c’ character is obtained as ‘A’ + ‘z’-’c’

Solution

```
#define MAXR 30
int main(void) {
  char ch, namefile[MAXR+1];
  FILE *fpin, *fpout;

  printf("name of input file: ");
  scanf(“%s”, namefile);
  fpin = fopen(namefile, "r");
  printf("name of output file: ");
  scanf("%s", namefile);
  fpout = fopen(namefile, "w");
```

```
  while (fscanf(fpin, "%c", &ch) == 1) {
    if (ch>= '0' && ch<= '9')
      ch = '0' +('9' -ch);
    else if (ch>= 'a' && ch<= 'z')
      ch = 'A' +('z' -ch);
    else if (ch>= 'A' && ch<= 'Z')
      ch = 'a' +('Z' -ch);
    fprintf(fpout, "%c",ch);
  }
  fclose(fpin); fclose(fpout);
}
```

Text-processing problems

- Problems where we need to manipulate sequences of characters and/or strings.
  - Example: modify a test, create a message in a specific format
- Possible purposes:
  - Text interpretation (input: menu with input of commands)
  - Text visualization (output: «elementary» graphics)
  - Text processing (modification: formatting)
- Operations can be:
  - At the single character level (lower level)
  - At the string level (higher level: a string is a whole word/sentence).

Text-processing problems

- C allows I/O both at the character and at the string level
- To manipulate text at the string level we need functions (either libraries or functions written by us).
- Comparison between texts:
  - At the character level, using relational operators (==, !=, >, <, >=, <=)
  - At the level of string, using the function strcmp (or strncmp)
  - You cannot use == or != with strings!!!
- Construct if:
  - The most general, it can always be used (switch can always be replaced by if, not viceversa)

Text-processing problems

- Switch makes a selection of different cases based on constant integer values. It can NEVER be used:
  - With strings (we would compare pointers, and not content of the strings)

```
switch (country) {
   case "Italy": ...
   ...
}
```

```
if strcmp (country, "Italy") == 0) {
   ...
}
else if (...)
...
```

  - With sets or intervals of characters (unless we enumerate each possible character one by one)

```
switch (car) {
   case 'A' : case 'B' : case 'C' : ...
   ...
}
```

```
if (car>= 'A' && car<= 'Z') {
   ...
}
else if (...)
...
```

Text-processing problems

- Switch makes a selection of different cases based on constant integer values. It can NEV
  - With strings (we would co

```
switch (country) {
  case "Italy": ...
  ...
}
```

We can use library functions (`ctype.h`): **`isalpha`**, **`isupper`**, **`islower`**, **`isdigit`**, **`isalnum`**, **`isxdigit`**, **`ispunct`**, **`isgraph`**, **`isprint`**, **`isspace`**, **`iscntrl`**

  - With sets or intervals of characters (unless we te each possible character one by one)

```
switch (car) {
  case 'A' : case 'B' : case 'C' : ...
  ...
}
```

```
if (car>= 'A' && car<= 'Z') {
  ...
}
else if (...)
...
```

Text-processing problems

- Switch makes a selection of different cases based on constant integer values. It can NEVER be used:
  - With strings (we would compare poin

```
switch (country) {
  case "Italy": ...
  ...
}
```

```
if str
  ...
}
else
...
```

```
if (isalpha(car)) {
   ...
}
else if (...)
...
```

  - With sets or intervals of characters (unless we enu ach possible character one by one)

```
switch (car) {
  case 'A' : case 'B' : case 'C' : ...
  ...
}
```

```
if (car>= 'A' && car<= 'Z') {
  ...
}
else if (...)
...
```

Functions on characters   #include <ctype.h>

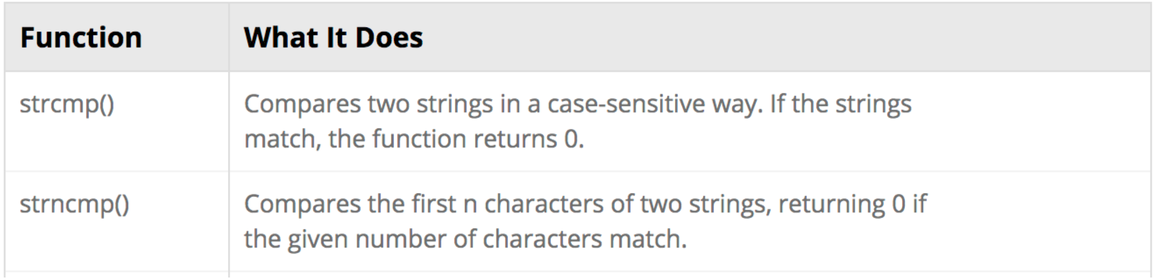
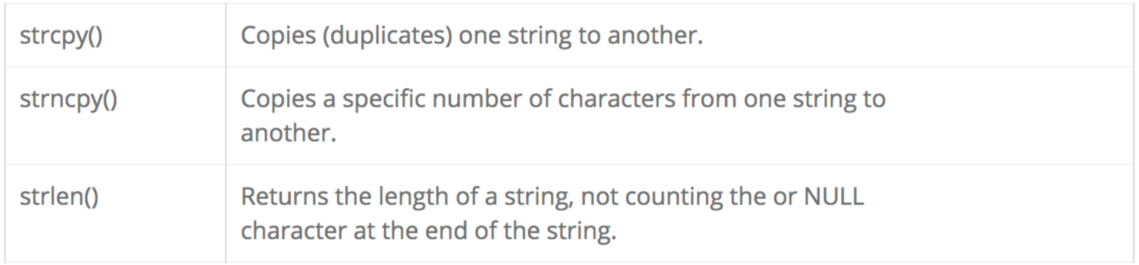
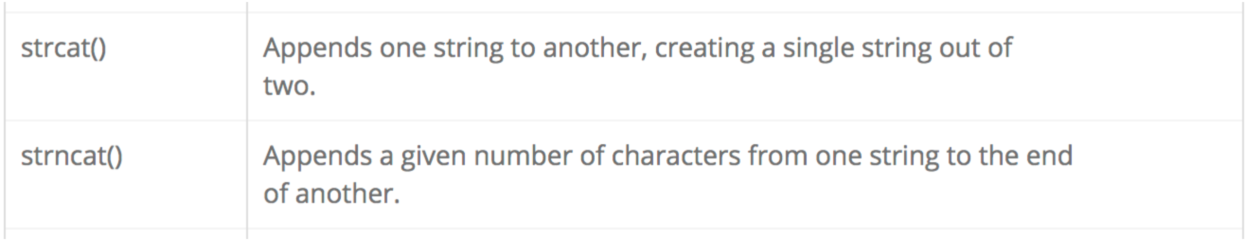
Function	**Returns TRUE When *ch* is**
isalnum(*ch*)	A letter of the alphabet (upper- or lowercase) or a number
isalpha(*ch*)	An upper- or lowercase letter of the alphabet
isascii(*ch*)	An ASCII value in the range of 0 through 127
isblank(*ch*)	A tab or space or another blank character
iscntrl(*ch*)	A control code character, values 0 through 31 and 127
isdigit(*ch*)	A character 0 through 9
isgraph(*ch*)	Any printable character except for the space
ishexnumber(*ch*)	Any hexadecimal digit, 0 through 9 or A through F (upper- or lowercase)
islower(*ch*)	A lowercase letter of the alphabet, *a* to *z*
isnumber(*ch*)	*See* isdigit()
isprint(*ch*)	Any character that can be displayed, including the space
ispunct(*ch*)	A punctuation symbol
isspace(*ch*)	A white-space character, space, tab, form feed, or an Enter, for example
isupper(*ch*)	An uppercase letter of the alphabet, *A* to *Z*
isxdigit(*ch*)	*See* ishexnumber()

Function	**Returns**
tolower(*ch*)	The lowercase of character *ch*
toupper(*ch*)	The uppercase of character *ch*

Strings

- They are not a specific data type, just a special vector of `char` elements
  - Example: `char nome[N];`
- They are characterize by a string terminator `'\0'` (ASCII code 0), placed after the last significant character:
  - Usually a vector is over-sized in the declaration, `'\0'` indicates where the string actually ends
- To perform operations on vectors of chars as a whole, it is necessary to use strings (with `'\0'`) as operands:
  - String constants (for example: "hello") are strings: they have a '\0’ terminator
  - IO functions: gets/puts, fgets/fputs, formatted IO with %s
  - Functions of the library <string.h>: for example, strlen, strcpy, strcmp, strncmp, strcat

Functions on strings   #include <string.h>

Function	What It Does
strcmp()	Compares two strings in a case-sensitive way. If the strings match, the function returns 0.
strncmp()	Compares the first n characters of two strings, returning 0 if the given number of characters match.
strcpy()	Copies (duplicates) one string to another.
strncpy()	Copies a specific number of characters from one string to another.
strlen()	Returns the length of a string, not counting the or NULL character at the end of the string.
strcat()	Appends one string to another, creating a single string out of two.
strncat()	Appends a given number of characters from one string to the end of another.

Strings

- It is also possible to handle strings as vectors of characters, and perform operations character-by-character
- Example: convert all the elements of a string to uppercase

```
…
int i;
char word[50];
scanf("%s",word);    // read word from keyboard

for (i=0; i<strlen(word); i++)
  word[i] = toupper(word[i]);
…
```

Selecting from a menu

- Selecting from a menu consists in making a choice among a finite set of available options to execute a task (a mathematical operation, an output message, a function call, etc.):
  - Typically, the choice is based on a textual information (a command/option)
  - The list of possible options should be printed on video (in case it is I/O based on keyboard/video)
  - We need to handle possible errors (or non-valid choices made by the user)
  - The menu is typically iterated in an infinite loop (one of the possible options terminates the program)

Menu with characters as selectors

- The simplest form. We foresee a finite number of options, each associated by a pre-defined constant character value (for example, the initial of a command)
- Typical complications:
  - We may need to discard a certain number of spaces before we are able to identify the selector character
  - If it is an alphabetic character, we may want to ignore the difference between uppercase and lowercase
- Solution: either if or switch constructs (most typically, switch)

Menu with characters as selectors

- Formulation:
  - Write a function that iteratively acquires a string from keyboard (50 characters maximum, may contain spaces)
    - The first character different from a space is the menu selector
    - If it is ‘E’ (exit), we need to terminate the iterations
    - If it is one of 'A', 'L', 'T’ (even lowercase), we need to call one of the respective functions fA, fL, fT, passing the rest of the string (**after** the selector) as argument
    - Any other character should trigger an error message

Solution

```
void menuCharacters (void) {
  const int MAXL=51;
  char row[MAXL];
  int i, toContinue=1;

  while (toContinue) {
    printf("command (A/L/T, E=exit): ");
    scanf(" %c", &sel);
    gets(row); /* rest of the line */

    ...
```

```
    ...
    switch (toupper(sel)) {
      case 'A' : fA(row); break;
      case 'L' : fL(row); break;
      case 'T' : fT(row); break;
      case 'E' : toContinue=0; break;
      default: printf("non-valid command\n");
    }
  }
}
```

Menu with words as selectors

- More complicated. We have a finite number of options, each selected by a distinct constant string: for example, the first word of a command
- Typical complications:
  - We may need to discard a few spaces before we identify the word that acts as the selector
  - We may want to ignore the difference between uppercase/lowercase alphabetic characters
- Solution: if construct (switch cannot be used with strings)

Menu with words as selectors

- Formulation:
  - Write a function that iteratively acquires a string from keyboard (50 characters at most, may contain spaces)
    - The first word different from space is the selector
    - If word is "end", we need to terminate the iterations
    - If the word is one among "seek", "mod", "pr" (case insensitive), we should call the respective functions seek, modify, print, passing the rest of the string (other than the selector) as the argument
    - Any other word should trigger an error message

Solution

```
void menuWord (void){
  const int MAXL=51;
  char command[MAXL], row[MAXL];
  int i, toContinue=1;
  while (toContinue) {
    printf(”command (seek/mod/pr/end): ");
    scanf("%s", command); /* command */
    for (i=0; i<strlen(command); i++)
      command[i] = toupper(command[i]);
    gets(row); /* rest of the line */
    ...
```

```
    ...
    if (strcmp(command, "SEEK")==0) {
      seek(row);
    } else if (strcmp(command, "MOD")==0) {
      modify(row);
    } else if (strcmp(command, "PR")==0) {
      print(row);
    } else if (strcmp(command, "END")==0) {
     toContinue =0;
    } else {
      printf("non valid command\n");
    }
  }
}
```

Text processing – at the character level

- A text is built/modified at the character level if:
  - the operation needs to happen character by character. There are no other options
  - The task may be executed at the string level, but it becomes more complicated. Easier at the character level, because we can use variants of available functions or we can handle particular cases in an easier way
- Caveat! If we manipulate strings character by character, we should not forget about the string terminator ('\0')

Producing elementary graphs/figures

- Figures composed by characters (the stdout is a matrix of 25 by 80 characters)
- It is NOT a pixel-wise graphics: for that, we need specific libraries (and languages)
- The visualization of characters on video (or text file) is sequential (left to right, up to down)
- We can visualize figures on video:
  - Printing directly the characters, row by row (examples in the following)
  - Using a matrix of characters (we will see examples in the next lectures).

Visualization of a rectangle

- Formulation:
  - Write a function that receives two integer numbers as arguments (b and h) and prints a rectangle of bxh ‘*’ characters on video
  - For example, if b=5 and h=4, the function should print:

```
*****
*   *
*   *
*****
```

- Algorithm: We just need nested loops to handle the visualization of the characters, row by row.

Solution

```
void rectangle (int b, int h) {
  int i, j;

  for (i=0; i<h; i++) {          // iterates on rows
    for (j=0; j<b; j++)          // iterates on columns
      if (i!=0 && i!=h-1 && j!=0 && j!=b-1)
        printf(" ");
      else
        printf("*");
    printf("\n");
  }
}
```

```
b=5, h=4

*****
*   *
*   *
*****
```

Visualization of a parabola

- Formulation:
  - Given the parabola of equation:

    $$y = ax^2 + bx + c$$
  - Write a program that:
    - Reads coefficients a, b, c from keyboard
    - Reads from keyboard an integer n (n>0), and the values (x0, xn), extrema of the x-axis interval
    - Reads from keyboard the values (ymin, ymax), extrema of the y-axis interval
    - Divide the [x0,xn] interval into n equal sub-intervals, delimitated by respectively x0,x1,x2,…,xn
  - Compute values y(xi) per each xi = x0..xn
  - Print on file a graph (with vertical X-axis) representing the parabola function in the rectangular portion of the plane identified by the intervals [x0,xn], [ymin,ymax], with rows representing y values and columns representing x values.

Visualization of a parabola

- Example: in case the values read from keyboard are the following
  - a=1.0, b=2.0, c=1.0, n=5, x0=0.0, xn=5.0, ymin=0.0, ymax=50.0
- We would obtain
  - y(0.0)=1.0, y(1.0)=4.0, y(2.0)=9.0, y(3.0)=16.0, x(4.0)=25.0, x(5.0)=36.0
  - The content of the file should be:

```
*
   *
        *
               *
                        *
                                   *
```

Visualization of a parabola

- Data structures:
  - FILE pointer (fpout)
  - coefficients: a, b, c (float)
  - Number of intervals: n (int)
  - Intervals: x0, xn, ymin, ymax (float)
  - Intermediate information: step (length of n intervals) x, y (float)
  - Counters to draw the graph: i, j (int)
- Algorithm:
  - input of variables and compute the step (= length of the intervals)
  - Iteration on x=x0..xn increased by step
    - compute y(x)
    - If it is within the interval [ymin,ymax], convert to integer (j) and print a ‘*’ after j spaces

Solution

```c
#include <stdio.h>
#include <math.h>
int main(void) {
  float a,b,c,x,step,x0,xn,y,ymin,ymax;
  int i, j, n;
  FILE *fpout = fopen("out.txt", "w");
  printf("Coefficients (a b c): ");
  scanf("%f%f%f",&a,&b,&c);
  printf(”Number of intervals: ");
  scanf("%d",&n);
  printf("Interval for x-axis: ");
  scanf("%f%f",&x0,&xn);
  printf("Interval for y-axis: ");
  scanf("%f%f",&ymin,&ymax);
  ...
```

```c
 ...
  step = (xn-x0)/n;
  for (i=0; i<=n; i++) {
    x = x0 + i*step;
    y = a*x*x + b*x + c;
    if (y>=ymin && y<=ymax) {
      for (j=round(y-ymin); j>0; j--)
        fprintf(fpout," ");
      fprintf(fpout,"*");
    }
    fprintf(fpout,"\n");
  }
  fclose(fpout);
}
```

Text processing – at the string level

- A text is processed at the string level if
  - It is possible to identify substrings where to apply operations in one shot
  - We want to use library functions for string processing, or use functions written ad-hoc by the programmer
- Typically, the substrings have spaces as separators (input is easy!)
- In other cases, the substrings may contain spaces or be delimited by other separator characters.

Text formatting

- Formulation:
  - A text file is given, that can be seen as a sequence of lines. Each line can be decomposed into substrings (words) of no more than 20 characters each, with spaces (or '\t' or '\n') as separators.
  - Write a C function that reads the file, copies the content to another file (the names of the two files are received as arguments) after performing the following formatting operations:
    - Sequences of spaces should be replaced by a single space
    - Newline characters ('\n') should be either substituted to the spaces or eliminated, so that each line has the largest possible length <=lmax (third argument passed to the function). Words should not be truncated.

Text formatting

- Solution:
  - Can be either at the character or at the string level
  - A possible solution to handle strings is based on the fact that fscanf("%s") allows to automatically isolate strings with spaces (or '\t’ or '\n’) as separators
    - Iteration of input of strings, and contextual computation of the length of a line
    - Before printing the string, based on the computation of the length of the line we decide if we need to print a newline character

Solution

```
void formatting (char nin[], char nout[],
                 int lmax) {
  const int STRLEN=21;
  FILE *fin=fopen(nin, "r");
  FILE *fout=fopen(nout, "w");
  char word[STRLEN];
  int l;

  l=0;
  ...
```

```
  while (fscanf(fin, "%s",word)==1) {
    if (l+1+strlen(word) > lmax) {
      fprintf(fout, "\n%s",word);
      l=strlen(word);
    }
    else {
        if (l!=0)
            fprintf(fout," ");
        fprintf(fout,"%s",word);
        l+=1+strlen(word);
    }
  }
  fclose(fin); fclose(fout);
}
```

Solution

```
void formatting (char nin[], char nout[],
                 int lmax) {
  const int STRLEN=21;
  FILE *fin=fopen(nin, "r");
  FILE *fout=fopen(nout, "w");
  char word[STRLEN];
  int l;
```

Test to avoid to print a space before
The first word

```
  while (fscanf(fin, "%s",word)==1) {
    if (l+1+strlen(word) > lmax) {
      fprintf(fout, "\n%s",word);
      l=strlen(word);
    }
    else {
        if (l!=0)
            fprintf(fout," ");
        fprintf(fout,"%s",word);
        l+=1+strlen(word);
    }
  }
  fclose(fin); fclose(fout);
}
```

Solution

```
void formatting (char nin[], char nout[],
                 int lmax) {
  const int STRLEN=21;
  FILE *fin=fopen(nin, "r");
  FILE *fout=fopen(nout, "w");
  char word[STRLEN];
  int l;

  l=0;
  ...
```

```
  while (fscanf(fin, "%s",word)==1) {
    if (l+1+strlen(word) > lmax) {
      fprintf(fout, "\n%s",word);
      l=strlen(word);
    }
    else {
      fprintf(fout,"%s%s",
                   l==0? "":" ",word);
      l+=1+strlen(word);
    }
  }
  fclose(fin); fclose(fout);
}
```

Solution

```
void formatting (char nin[], char nout[],
                 int lmax) {
  const int STRLEN=21;
  FILE *fin=fopen(nin, "r");
  FILE *fout=fopen(nout, "w");
  char word[STRLEN];
  int l;
```

```
  while (fscanf(fin, "%s",word)==1) {
    if (l+1+strlen(word) > lmax) {
      fprintf(fout, "\n%s",word);
      l=strlen(word);
    }
    else {
      fprintf(fout,"%s%s",
                    l==0? "":" ",word);
      l+=1+strlen(word);
    }
  }
  fclose(fin); fclose(fout);
}
```

Test to avoid to print a space before
The first word

Verification and selection problems

- Verification: decide whether a set of information or data are compliant with specific acceptance criteria :
  - The answer is Boolean (YES/NO)
  - It can involve single variables or sequences of data
  - It can involve one or multiple verifications (on different data)
- Selection: distinguish data based on acceptance/verification criteria

Acceptance criteria

∀, ∃ Gottlob Frege, 1879

Criteria can be

- Logic expressions (propositional logic)
- Conditions (property) p on a set of data S, expressed by:
  - Universal quantifier ∀:

    $\forall x \in S \mid p$ is true

    per each x that belongs to S property p is true
  - Existential quantifier ∃:

    $\exists x \in S \mid p$ is true

    There exists at least one x that belongs to S so that property p is true

Duality $\forall$, $\exists$ ($\neg$ logic negation operator)

- $\neg(\forall\ x \in S \mid p \text{ is true}) \Leftrightarrow \exists\ x \in S \mid p \text{ is false}$

  It is not true that p is true per each x that belongs to S

  IS LIKE SAYING

  there exists at least one x belonging to S so that p is false

- $\neg(\exists\ x \in S \mid p \text{ is true}) \Leftrightarrow \forall\ x \in S \mid p \text{ is false}$

  It is not true that there exists at least one x belonging to S so that p is true

  IS LIKE SAYING

  Per each x that belongs to S, p is false

Example: monotonicity of a sequence

Given a sequence of N integers S = (x0, x1, … , xN-1):

- property p: S is monotonously increasing

  $\forall$ xi, xi+1 $\in$S | (xi $\leq$ xi+1) 0 $\leq$ i < N-1

  Per each pair of subsequent elements, there is a ≤ relation

- property $\neg$ p: S is NOT monotonously increasing

  $\exists$ xi, xi+1 $\in$S | (xi > xi+1) 0 $\leq$ i < N-1

  There exists at least one pair of subsequent elements in S so that the relation ≤ is not true (there is a > relation)

Monotonicity of a sequence

- C implementation
- Integer variable used as a representation logic of the universal quantifier, initialized to 1
- Iterative construct to enumerate all the elements of S
- Per each iteration:
  - check if the initial hypothesis (property holds) is verified
  - the logic variable is modified ONLY if the initial hypothesis is NOT verified

Solution

```
...
int Monotonous = 1; // property holds: initialized to 1
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi);

for (i=1; i<N; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if (xi > xj)  // test on subsequent values
    Monotonous = 0;  // update the logic variable
  xi = xj;
}
```

Common mistake

```
int Monotonous = 1;
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi);
for (i=1; i<N; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if(xi > xj)
    Monotonous = 0;
  else
    Monotonous = 1;
  xi = xj;
}
```

The result will only depend on the last comparison

Improvement: early exit (structured)

```
...
int Monotonous = 1;
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi

for (i=1; i<N && Monotonous ==1 ; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if(xi > xj)
    Monotonous = 0;
  xi = xj;
}
```

Exit from loop using flag

Variant: early exit - not structured (1)

```
int Monotonous = 1;
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi);
for (i=1; i<N; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if(xi > xj) {
    Monotonous = 0;
    break;
  }
  xi = xj;
}
```

Exit from loop with break

Variant: early exit - not structured (2)

```
int Monotonous = 1;
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi);
for (i=1; i<N; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if(xi > xj) {
    Monotonous = 0;
    return 0;
  }
  xi = xj;
}
return 1;
```

Non-stuctured early exit from function

Variant: early exit - not structured (2)

```
~~int Monotonous = 1;~~
int i;
int xi, xj;
printf("x0= "); scanf("%d", &xi);
for (i=1; i<N; i++) {
  printf("x%d= ", i); scanf("%d", &xj);
  if(xi > xj) {
    ~~Monotonous = 0;~~
    return 0;
  }
  xi = xj;
}
return 1;
```

The logic variable is Useless in this case

Verification of sequences

- To verify a sequence means to check whether the sequence is compliant with the acceptance criteria
- Examples of criteria:
  - sum/average of data >= or <= a given limit
  - Comparison of adjacent data: increasing, difference <= threshold, etc.
  - Rules on groups of adjacent data:
    - There should be no more than 2 consecutive consonants, a punctuation sign should be followed by a space, 'p’ and 'b’ cannot be preceded by 'n’ …
    - A certain transmitted data packet should comply with specific formatting rules.

Verify alphabetical order

- Formulation:
  - A text file contains a list of people, one per line (50 chars max), represented by surname and name (they may contain spaces)
  - Write a C function that receives the file pointer (already open) as argument, checks whether the data are in alphabetical order. It should return 1 if YES, 0 if NO
- Solution:
  - We should iteratively analyze data. At each iteration we should compare the last two data acquired
  - Acceptance : quantification
  - Verdict: value returned by the function
  - Let’s try an early exit (non-structured)

Verify alphabetical order

- Data structures:
  - Two strings, respectively for the last line (line1) acquired from the file and for the line that was acquired before that (line0)
- Algorithm:
  - Iteratively reads lines from the file and compares the last two:
    - Whenever the ordering criteria is violated, we should return 0
    - If the ordering is not violated, we proceed
    - At the end of each iteration, line1 becomes line0 and a new line1 is acquired
    - If the ordering is never violated, we return 1 at the end of the function
  - The acquisition of the first line needs to be handled separately

Solution

```
int verifyOrder (FILE * fp) {
  const int MAXC=50;
  char line0[MAXC+1], line1[MAXC+1];

  fgets(line0,MAXC,fp);  // reads first line separately
  while (fgets(line1,MAXC,fp)!=NULL) {
    if (strcmp(line1,line0)<0)  // if line1 precedes line0
      return 0;
    strcpy(line0,line1);  // copies line1 in line0
  }
  return 1;
}
```

Verify data consistency

- Formulation:
  - A text file contains a sequence of temperatures (Celsius degrees) detected by a thermal sensor within a day, with a 5 minutes interval. Temperatures are real numbers separated by either spaces or '\n' characters
  - Write a C function that receives the file pointer (already open) as argument and checks that the temperature does not vary of more than 5 degrees in each 10-minutes interval. The function returns 1 if the temperatures are compliant with the criteria, 0 otherwise

Verify data consistency

- Solution:
  - Iteratively analyze data, checking each time the latest three temperature values (3 values cover a 10 minutes interval)
  - Acceptance criteria:
    - Boolean condition on first 2 values
    - quantification on remaining values
  - Verdict: value returned by the function
  - Early exit (non-structured).

Verify data consistency

- Data structure:
  - 3 real variables (float) for the last three readings: t0, t1, t2 (t2 is the last reading)
- Algorithm:
  - Iteratively acquire readings, checking the last three (first two values should be acquired and checked separately at the beginning)
    - As long as |t2-t0|>5 OR |t2-t1|>5 (|t1-t0|>5 is already checked), return 0
    - Otherwise, we update t0 and t1 and acquire next reading
    - If the criteria is never violated, return 1 at the end of the function.

Solution

```
int verifyTemperature (FILE *fp) {
  float t0, t1, t2;
  fscanf(fp, "%f%f",&t0,&t1);
  if (abs(t1-t0)>5)
    return 0;           // if first two values do not comply, early exit
  while (fscanf(fp, "%f",&t2)==1) {
    if (abs(t2-t0)>5 || abs(t2-t1)>5)    // || is logical OR
      return 0;
    t0=t1;
    t1=t2;
  }
  return 1;
}
```

Data selection

- While verifying data or sets of data, it is possible to distinguish between categories of data based on the acceptance criteria
- Selection in this case is a variant of verification:
  - Data are first verified
  - The ones corresponding to the criteria, are selected
- Typically, iterative process:
  - Multiple data (or sets of data) are verified at each iteration
  - A portion of the data is selected

Study grants

- Formulation:
  - A text file contains a list of student, as follows:
    - A line contains the matricola id (preceded by ‘#’)
    - Following line contains surname and name
    - Following lines report the scores obtained in each exam, one per line
  - Write a C program that
    - Reads the file and acquires from keyboard two numbers: mmin (real) and nmin (integer)
    - Selects the students whose average score is not lesser mmin and whose number of exams is not lesser than nmin
    - Writes names and matricola IDs of the selected students on a second file (both file names are read from keyboard).

Study grants

- Solution:
  - Iteratively analyze the data, considering each time the information of a single student. Per each of them, we need to compute number of exams and average score
- Data structure:
  - 2 strings for names of files, surname and name and matricola: s0, s1
  - 2 file pointers (files should be opened at the same time, one in read and the other in write mode) : fin, fout
  - 2 integers for the exams count (ne) and minimum count (nmin)
  - 3 floats for current score (score), average score (avg) and minimum average (mmin)

Study grants

- Algorithm:
  - Data verification consists in:
    - Reading data corresponding to one student
    - Computing number of exams and average score
    - Comparing computed values with minimum thresholds
  - The solution consists into a nested loop:
    - External loop (per each student): read matricola, surname and name
    - Internal (per each score): update sum of scores and exam counting –> compute average score
  - The separation between two following students is based on recognizing a '#' character (it starts a new section corresponding to the next student).

Solution

```
#define MAXL 100
#include <stdio.h>
int main(void) {
  char s0[MAXL+1], s1[MAXL+1];
  FILE *fin, *fout;
  int nmin, ne;
  float mmin, avg, score;

  printf("File input: "); scanf("%s",s0);
  printf("File output: "); scanf("%s",s1);
  fin = fopen(s0, "r"); fout = fopen(s1,"w");
  printf(”thresholds min. n. exams and avg score:");
  scanf("%d%f", &nmin, &mmin);
  ...
```

```
  while (fgets(s0,MAXL,fin)!=NULL) {
    fgets(s1,MAXL,fin);
    ne=0; avg=0.0;
    while (fscanf(fin, "%f",&score)==1) {
          avg += score; ne++;
    }
    avg = avg /ne;
    if (avg >= mmin && ne>=nmin) {
      fprintf(fout, "%s%s",s0,s1);
    }
  }
  fclose(fin); fclose(fout);
}
```

Solution

```c
#define MAXL 100
#include <stdio.h>
int main(void) {
  char s0[MAXL+1], s1[MAXL+1];
  FILE *fin, *fout;
  int nmin, ne;
  float mmin, avg, score;

  printf("File input: "); scanf("%s",s0);
  printf("File output: ")
  fin = fopen(s0, "r"); f
  printf(”thresholds min.
  scanf("%d%f", &nmin, &mmin);
  ...
```

```c
  while (fgets(s0,MAXL,fin)!=NULL) {
    fgets(s1,MAXL,fin
    ne=0; avg=0.0;
    while (fscanf(fi          ,&score)==1) {
          avg += sco
    }
    avg = avg /ne;
    if (avg >= mmi

  fclose(fin); fclose(fout);
}
```

Reads a line containing matricola ID (without verifying the presence of ‘#’)

Solution

```
#define MAXL 100
#include <stdio.h>
int main(void) {
  char s0[MAXL+1], s1[MAXL+1];
  FILE *fin, *fout;
  int nmin, ne;
  float mmin, avg, score;

  printf("File input: ")
  printf("File output: "
  fin = fopen(s0, "r"); fout = fopen(s1,"w");
  printf(”thresholds min. n. exams and avg score:");
  scanf("%d%f", &nmin, &mmin);
  ...
```

```
  while (fgets(s0,MAXL,fin)!=NULL) {
    fgets(s1,MAXL,fin);
    ne=0;  vg=0.0;
    whi   (fscanf(fin, "%f",&score)==1) {
          avg += score; ne++;

          avg /ne;
              min && ne>=nmin) {
          ut, "%s%s",s0,s1);

  }
  fclose(fin); fclose(fout);
}
```

Reads name and surname

Solution

```
#define MAXL 100
#include <stdio.h>
int main(void) {
  char s0[MAXL+1], s1[MAXL+1];
  FILE *fin, *fout;
  int nmin, ne;
  float mmin, avg, score;

  printf("File input: "); scanf("%s",s0);
  printf("File output: "); scanf("%s",s1);
  fin = fopen(s0, "r"
  printf(”thresholds
  scanf("%d%f", &nmin
  ...
```

```
  while (fgets(s0,MAXL,fin)!=NULL) {
    fgets(s1,MAXL,fin);
    ne=0; avg=0.0;
    while (fscanf(fin, "%f",&score)==1) {
          avg += score; ne  ;
    }
    avg = avg /ne;
    if (avg >=              >=nmin) {
      fpri                 ",s0,s1);
}
```

Acquires numbers, until the ‘#’ stops the acquisition

Sorting problem

- A sorting problem consists in the request of permutating a sequence of data so that it verifies a speficic ordering criteria
- To sort data, we need an operator (or a function) that compares pairs of data, so that we are able to decide which of the two elements precedes the other based on the ordering criteria (Data equality is also admitted, so that any of the two can precede the other). Typically, based on relational operators (<, <=, >, >=).

Total or partial sorting

- "Total" sorting:
  - Based on the chosen criteria, one element will precede all the following ones in the sorted sequence. Each element can be compared with all others.
  - Example: alphabetical sorting of names
- "Partial" sorting:
  - The comparison criteria does not define a precedence for some pairs of elements.
  - Examples:
    - Alphabetical sorting of lists of names and numbers (alphabetical order does not apply to numbers);
    - Sorting names based on initial (Relative order between names with same initial is not defined)

Partial sorting algorithms

- The majority of algorithms for "total" sorting requires the use of arrays for intermediate steps. We will deal with them in the lectures dedicated to arrays.
- Simple “partial” sorting can be obtained by iterative selection
- As we will need to analyze data multiple times, to avoid the use of arrays we will make use of a file that will be read multiple times. It is a **quite unefficient solution**, just to avoid arrays **for the time being**.

Sorting points based on quadrant

- Formulation:
  - A text file contains a list of cartesian points, one per line, each represented by its two real coordinates
  - Write a C function that re-writes the points on a second file (the name of both files are received as parameters), after sorting them based on the quadrant they belong to
  - Point belonging to the X and Y axis can be assigned to any of the two adjacent quadrants they belong to

Sorting points based on quadrant

- Solution:
  - Read input file 4 times, each selecting (and printing on the output file) the points belonging to the quadrant of the corresponding iteration
- Data structure and algorithm:
  - It is opportune to implement a function that selects data of a quadrant. This function is called 4 times. Each quadrant is identified by two integer parameters, corresponding to X and Y respectively: +1 represents positive values, -1 negative values. Hence, the 4 quadrants are represented by the pairs (1,1), (-1,1), (-1,-1), (1,-1).

Solution

```c
void sortPoints (char nin[], char nout[]) {
  FILE *fin, *fout;
  fout = fopen(nout, "w");
  fin = fopen(nin, "r");
  selectPoints(fin,fout,1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,-1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,1,-1);
  fclose(fin);
  fclose(fout);
}
```

```c
void selectPoints (
  FILE *fi, FILE *fo, int sx, int sy) {
  float x,y;
  int xOK, yOK;
  while (fscanf(fi, "%f%f",&x,&y)==2) {
    xOK = x*sx>0.0 || (x==0.0 && sx>0);
    yOK = y*sy>0.0 || (y==0.0 && sy>0);
    if (xOK && yOK)
      fprintf(fo, "%f %f\n",x,y);
  }
}
```

Solution

```
void sortPoints (char nin[], char nout[]) {
```

X (Y) OK
if concordant with sx(sy)

```
  selectPoints(fin,fout,1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,-1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,1,-1);
  fclose(fin);
  fclose(fout);
}
```

```
void selectPoints (
  FILE *fi, FILE *fo, int sx, int sy) {
  float x,y;
  int xOK, yOK;
    ile (fscanf(fi, "%f%f",&x,&y)==2) {
    xOK = x*sx>0.0 || (x==0.0 && sx>0);
    yOK = y*sy>0.0 || (y==0.0 && sy>0);
    if (xOK && yOK)
      fprintf(fo, "%f %f\n",x,y);
  }
}
```

Solution

```
void sortPoints (char nin[], char nout[]) {
```

```
  selectPoints(fin,fout,1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,-1,-1);
  fclose(fin); fin = fopen(nin, "r");
  selectPoints(fin,fout,1,-1);
  fclose(fin);
  fclose(fout);
}
```

X (Y) OK
if concordant with sx(sy)

```
void selectPoints (
  FILE *fi, FILE *fo, int sx, int sy) {
  float x,y;
  int xOK, yOK;
    ile (fscanf(fi, "%f%f",&x,&y)==2) {
    xOK = x*sx>0.0 || (x==0.0 && sx>0);
    yOK = y*sy>0.0 || (y==0.0 && sy>0);
    if (xOK && yOK)
      fprintf(fo, "
```

X(Y) OK if ==0 and sx(sy) positive (conventionally 0 is interpreted as positive)