Programming and Algorithms

C: Strings; Cache memory.

Jean-Bernard Hayet

CIMAT, A.C.

Formatted input/output

Formatted input/output

Functions to write/read data following a specific format:

```
int  fprintf(FILE *stream, const char *format, ...);
int  fscanf(FILE *stream, const char *format, ...);
```

Data specifiers

- i,d: integers;
- u: unsigned;
- f: floating numbers;
- e: floating numbers in exponential notation;
- x,o: hexadecimal and octal;
- s: string;
- c: character;
- lf,ld, lld: long precision (for scanf).

Integer input/output

"%wd"

- w is the minimal width of number to read/print.
- With output, in case of having less digits than the specified number, then **spaces** are added.
- With input, in case of having more digits than the specified number, the read number is **truncated**.

```
int x = 67;
fprintf(stdout,"%5d\n",x);
```

```
   67
```

Float output

"%w.df"

- w is the minimum width of output data;
- d is the digits to be printed **after the decimal point**.

```
float x = 67.25;
fprintf(stdout,"%7.3f\n",x);
```

```
 67.250
```

Flags

0	Pad with zero instead of spaces.
-	Left justification (padding on the right).
+	To always print the sign.

Much more possibilities in C++.

Strings

Strings

- In general, encoded on 1 octet.
- **ASCII** characters: on 7 bits.
- **ISO-8859** characters: on 8 bits (**ISO-8859-1**: characters from Western languages).
- Example: 233 corresponds to "é"
- **Unicode**: another standard. with representations in series of 8, 16 and 32 bits (UTF-8,UTF-16,UTF-32).
- In C, Unicode needs to use a special type of character type (wchar_t) and special I/O functions.

Strings

From file streams:

```c
char* fgets(char *str, int size, FILE *stream);
```

The fgets() function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a character is appended to end the string.

Strings

```c
#include <stdio.h>
int main() {
        char test [10];
        fgets ( test ,10, stdin );
        fprintf ( stderr , "%s \n", test );
}
```

Gives:

```
jbhayet@Barkoxe:~$ ./a.out
Les sanglots longs des violons de l'automne
Les sangl
```

Strings

```c
int sprintf(char *cadena, const char *format , ...) ;
int sscanf(char *cadena, const char *format, ...) ;
```

Same syntax as fprintf. As any function on strings, it requires string.h.

Strings

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Compares strings through the lexicographic order. Returns a positive value if s1 > s2, 0 if they are equal, etc.

Can be applied either on the full string or on the first n characters.

Strings

```c
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t len);
```

Concatenates string s2 to string s1, and returns s1 (note the const modifier).

```c
char *strcpy(char *s1, const char *s2 );
char *strncpy(char *s1, const char *s2, size_t len);
```

Copy string s2 (including the terminating character) into s1.

Caution: With strncpy, when s2 is larger than s1, s1 will not have the terminating character.

Strings

If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there's enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.

Strings

```c
char *strchr( const char *s, int c);
```

Localizes the character c in the string and returns a pointer to its position.

```c
size_t strlen( const char *);
```

String size (before the terminating character).

What is size_t?

This is an **alias** for some of the unsigned integers.

The only specification in the language is that it is **at least 16 bits**.

It is used to represent the **size of objects**, in bytes: it is the type returned by sizeof or by strlen.

Strings

To split a large strings into tokens, according to specified delimiters:

```
char *strtok(char *str, const char *delim);
```

returns a pointer to the character of next token replaces delimiters by \0 next calls with NULL

Strings

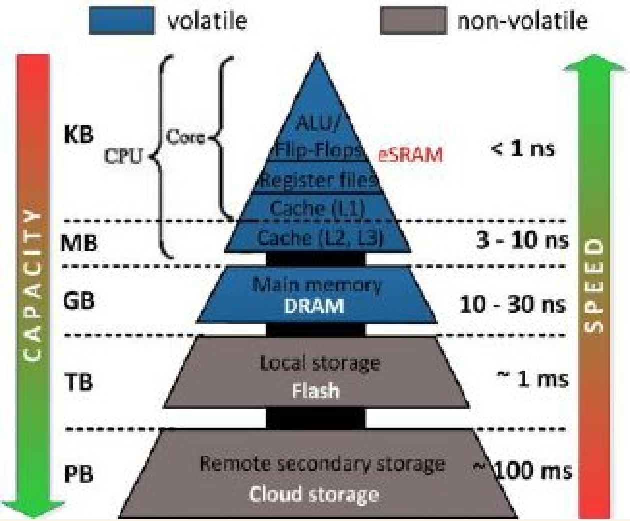
```
int main() {
        char str[] = "Les sanglots longs des violons de l'automne";
        char delimiter[] = " '";
        char *str_ptr = strtok(str, delim);
        while(str_ptr != NULL)
        {
                printf("%s\n", str_ptr);
                ptr = strtok(NULL, delimiter);
        }
        return 0;
}
```

Strings

```
jbhayet@Barkoxe:~/Dropbox/WorkCIMAT/Courses/Programació
Les
sanglots
longs
des
violons
de
l
automne
```

Cache memory

Memory hierarchy



From MTJ-based hybrid storage cells for "normally-off and instant-on" computing.

Memory hierarchy

Memory is organized **hierarchically**:

- The **time latencies** vary a lot!
- The most efficient memory spaces are also the **smallest**.
- When running a program, parts of the slowest memory spaces may be **copied for efficient usage** the the fastest memory spaces (example: RAM to Cache or RAM to CPU Registers).

Memory hierarchy

- RAM/Hard drive/Cloud storage.
- Within "**volatile**" memory:
    - CPU (registers/eSRAM)
    - Cache memory, with distinct levels (L1,L2,L3).

In some applications, the program efficiency can be improved dramatically by an efficient use of the cache memory.

Do not worry: **You do not have to do it by yourself**, but you have to give a little help to the compiler!

Locality

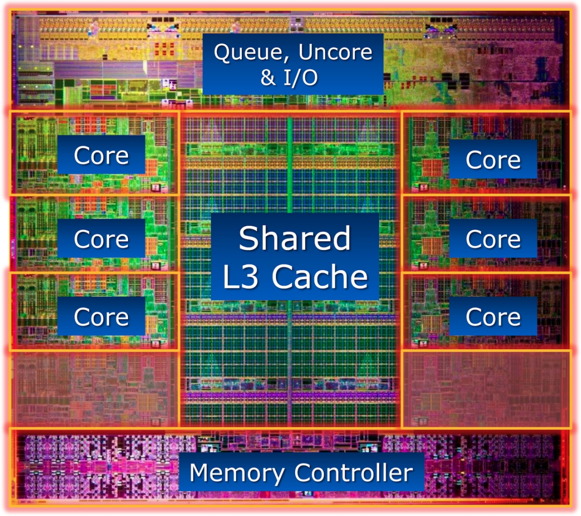
Typically, during the execution of a program, variables tend to be used **frequently** (repetition in **time**) and **together** (correlated use of **close** variables).

Temporal locality.

- Data currently read **may be re-read soon**.
- To make these re-readings **faster in the short term future**, as an **optimization**, one **can store a read data** in the faster cache memory to avoid searching again for the same data.

Spatial locality. Instructions/data close to the place in memory being used, may be needed in the near future.

Memory hierarchy



Intel

Cache memory: Access

- The copy from main memory to memory cache is done (to make it more efficient) through **blocks of contiguous memory**.
- The block is called the **cache line**.
- Typically the cache line is **64 bytes**.

Cache memory: Access

```
jbhayet@Barkoxe:~$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  9437184
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Cache memory: Access

During execution, when a variable (or instruction) needs to be accessed, the processor **first tries to locate it in the cache**.

Two possible outputs:

- the requested memory location is in the cache; in that case, the variable is read from the cache ("**cache hit**");
- the requested memory location is not in the cache; then a cache line is copied from the main memory ("**cache miss**").

Ideal situation: **many consecutive cache hits**.

Cache memory: Access

The performance of the use of the cache is measured through the hit ratio indicator:

$$hits/(hits + misses)$$

The closest to one, the better (the fastest the execution); the **organization of the code** is very important to improve this hit ratio.

Cache memory: Access

Example:

- Wen manipulating a **large matrix** (double pointer, static arrays, single pointer) through columns, **you may be quite inefficient**,
- The elements along columns are **probably far away in the main memory**.
- Each consecutive access may conclude in a cache miss and the copy into cache.
- When reading through **rows**, you take advantage of the previous cache hits to read the consecutive data directly from the cache.

Cache memory: Multiplication of matrices

```
for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
                for (int k=0;k<p;k++)
                        P[i][j] = P[i][j] + A[i][k] * B[k][j];
```

Correct? Efficient? Thoughts?

Cache memory: Multiplication of matrices

- Readings of A[i][k] are **mostly hits** (contiguous memory).
- Readings of B[k][j] are **mostly misses** (separate places in memory).
- Readings of P[i][j] can be set out of the inner loop.

Cache memory: Multiplication of matrices

Now consider:

```
for (int i=0;i<n;i++)
    for (int k=0;k<p;k++)
        for (int j=0;j<m;j++)
            P[i][j] = P[i][j] + A[i][k] * B[k][j];
```

Cache memory: Multiplication of matrices

- Readings of A[i][k] can be set out of the inner loop.
- Readings of B[k][j] and P[i][j] are **mostly hits** (the index, in both cases, corresponds to **rows**, not columns, i.e. contiguous places in memory).
- May be **way more efficient** when the size of the matrices is large.

Cache memory: Multiplication of matrices

```
#include <time.h>
#include <stdio.h>
#define n 100
#define m 5000
#define p 5000
double A[n][p],B[p][m],P[n][m];

int main() {

clock_t start = clock();
for (int c=0;c<10;c++)
        for (int i=0;i<n;i++)
                for (int j=0;j<m;j++) {
                        double p_v = 0.0;
                        double *a_ptr = A[i];
                        for (int k=0;k<p;k++) p_v += a_ptr[k] * B[k][j];
                        P[i][j] = p_v;
                }
clock_t end = clock();
double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Normal multiplication : %lf\n",cpu_time_used);
```

Cache memory: Multiplication of matrices

```
        start  = clock();
        for (int c=0;c<10;c++)
                for (int i=0;i<n;i++)
                        for (int k=0;k<p;k++) {
                                double a_v    = A[i][k];
                                double *b_ptr = B[k];
                                double *p_ptr = P[i];
                                for (int j=0;j<m;j++)
                                        p_ptr[j] += va * b_ptr[j];
                        }
end    = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
 printf("Improved multiplication : %lf\n",cpu_time_used);
```

Cache memory: Multiplication of matrices

```
jbhayet@Barkoxe:~/Dropbox/WorkCIMAT/Courses/Programación y A
Normal multiplication: 119.462736
Improved multiplication: 63.136964
```

Cache memory: Other examples

Given a big structure Data, which one is best?

Store values?

```
Data d_array[100];
```

or use the heap?

```
Data *d_array[100];
for (int i=0;i<100;i++)
        d_array[i]=(Data*)malloc(sizeof(Data));
```

Cache memory: Other examples

In the same context: Imagine you have an array of large structures where any of them will be discarded by a test:

```
for (int i=0; i < 1000; i++) {
        if (d_array[i].someFlag) {
                /* Do something */
                ...
        }
}
```

Improvement?