

Modulo 11

La pratica della programmazione

Laboratorio di Sistemi Operativi I
Anno Accademico 2007-2008

Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Come scrivere il codice

- **Obiettivo: codice che funziona, facile da leggere, correggere e mantenere**
- A cosa bisogna prestare attenzione (in ordine approssimativamente cronologico nel progetto):
 - la modularità e le interfacce
 - la scelta degli algoritmi e delle strutture dei dati
 - la notazione e lo stile
 - il debugging ed il collaudo
- Cosa si deve ricercare a tutti i costi:
 - ***semplicità*** - funzioni corte e semplici da gestire
 - ***chiarezza*** - sorgenti di facile interpretazione sia per l'utente che per il computer
 - ***generalità*** - moduli che funzionano in molteplici situazioni e che si adattano bene a situazioni impreviste

Obiettivi di progetto

- **Esplicitare chiaramente tutti gli obiettivi.**

Esempio da un caso reale:

The main technical objectives are:

- 1) the realization of an Imagery Management and Processing System able to demonstrate its relevance by means of:
 - the provision of parallel image processing algorithms
 - the flexible management (storage, retrieval, dissemination) of imagery data
- 2) the integration of a multi-tier software system based on both commercial packages and specialized newly developed modules for building up an end-to-end remote sensing data management system

La modularita'

La separazione in moduli indipendenti, come la separazione in procedure indipendenti, serve a limitare la complessita' dell'applicazione. Interventi successivi di modifica al sistema saranno tipicamente limitati ad una procedura o, al piu', ad un modulo.

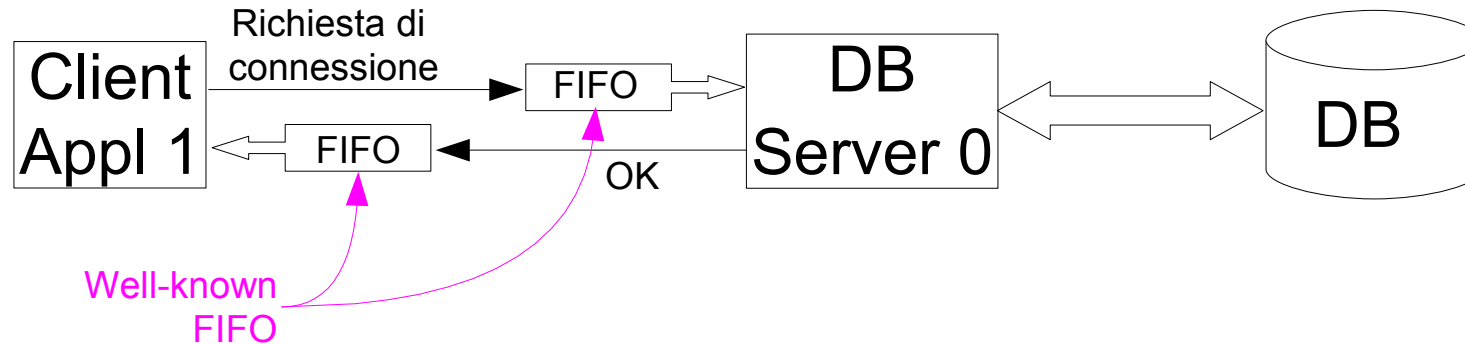
DIVIDE ET IMPERA

Cos'e un modulo?

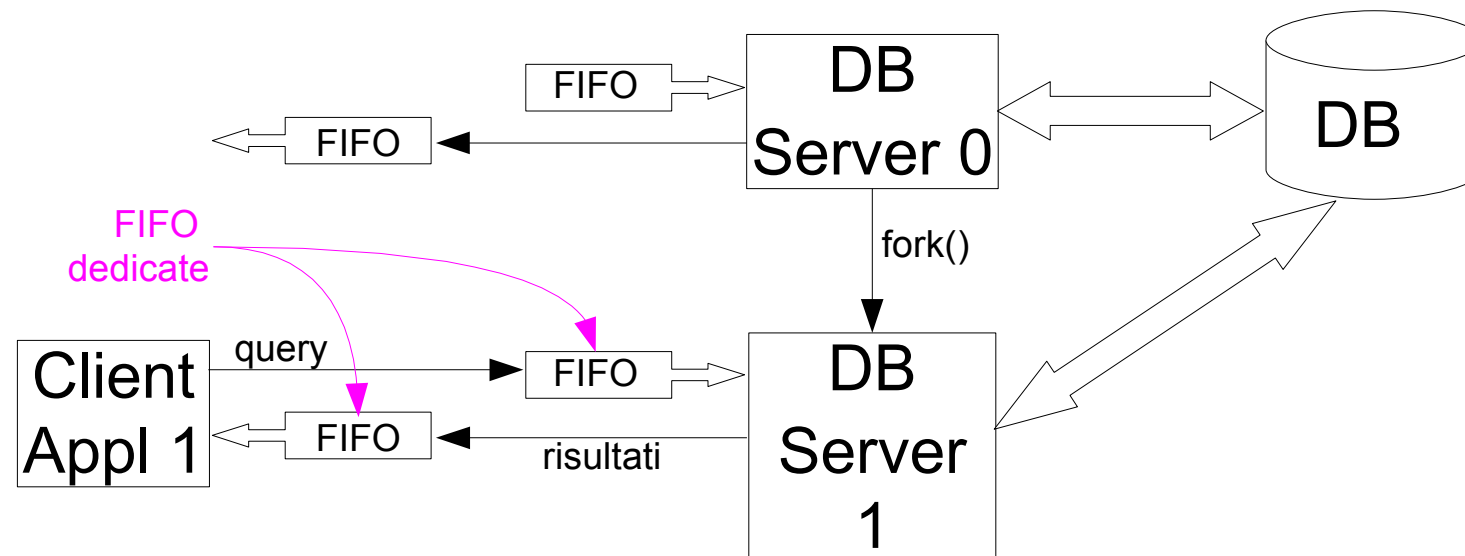
- ♦ I moduli sono strutture indipendenti che possono svolgere autonomamente un'importante funzione del sistema (per es., gestire l'interfaccia utente o l'accesso ai dati, la comunicazione tra processi o macchine, etc.)
- ♦ Possono essere implementati come librerie, processi o thread, anche con linguaggi diversi da quelli usati per altri moduli
- ♦ Un modulo di norma comprende piu' procedure con i loro dati (che sono quindi locali al modulo ma globali per le procedure che lo compongono)
- ♦ I moduli sono debolmente accoppiati tra loro tramite interfacce ben definite (protocolli di comunicazione, chiamate di libreria)
- ♦ La divisione in moduli, eventuali sottomoduli e procedure deve aiutare la *stratificazione* dell'applicazione, in cui ogni strato e' completo e consistente ed aggiunge dettagli allo strato soprastante nascondendo i dettagli dello strato sottostante

Moduli ed interfacce: esempio I

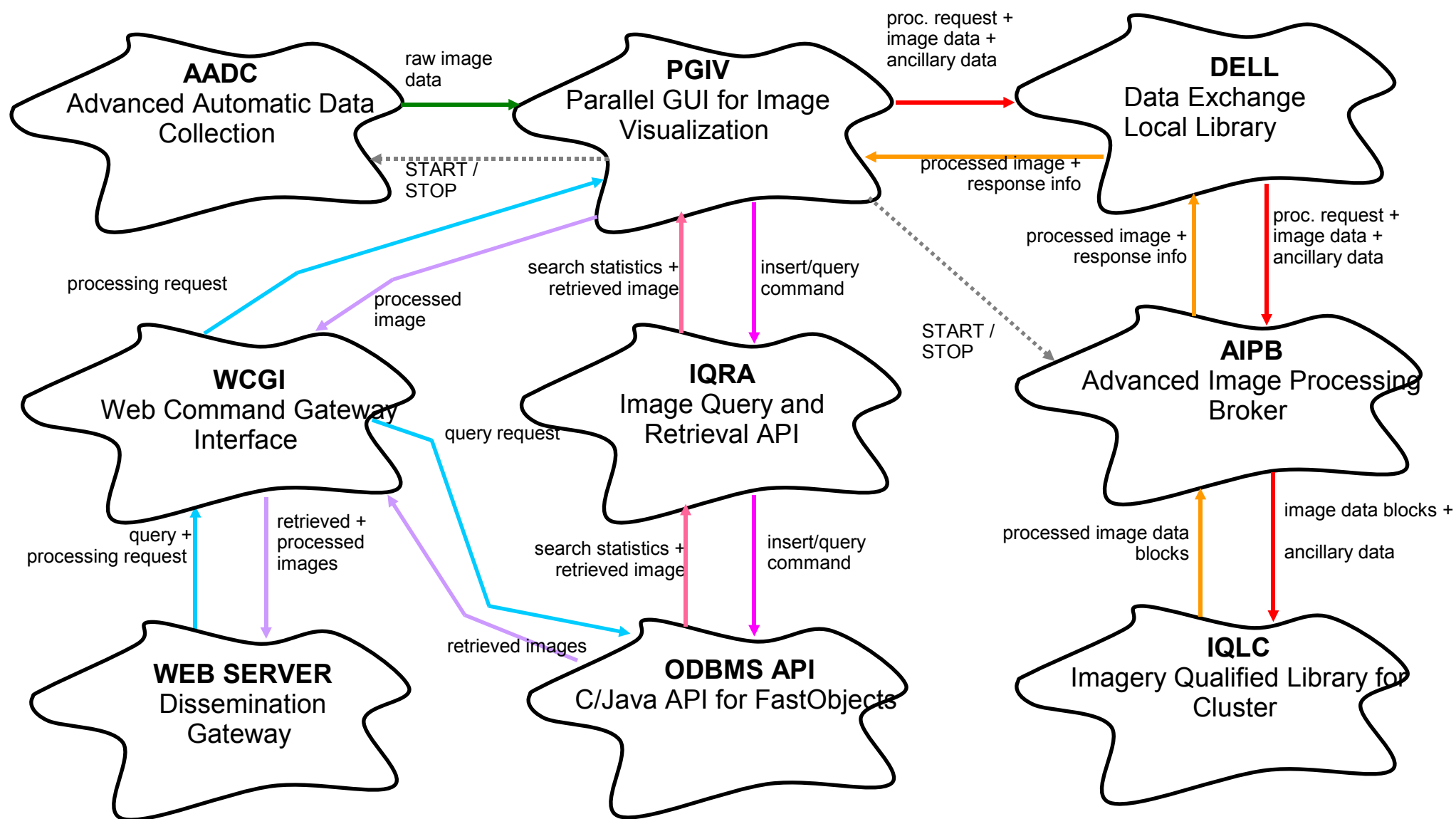
A) Connessione



B) Interazione col DB Server



Moduli ed interfacce: esempio II



Cos'e' una procedura?

- Una procedura e' la piu' piccola unita' indipendente di programmazione
- Tipicamente implementa un intero algoritmo od una funzione autoconsistente (e.g., lettura di un record da un file)
- Una procedura...
 - Deve essere breve (1-2 schermate)
 - Più e' complessa, più breve deve essere
 - Deve usare poche variabili (max 10)
 - Deve avere un solo punto di uscita
 - Deve essere ben commentata

Come si definisce un'interfaccia - I

- ♦ Quali servizi devono essere forniti?
 - ♦ definire chiaramente gli scopi ed i limiti del modulo
 - ♦ scopi e limiti debbono essere chiari e consistenti (e.g., gestione di file)
 - ♦ l'interfaccia e' un *contratto* che formalizza scopi e limiti del modulo
- ♦ Come si definiscono le specifiche?
 - ♦ scegliere poche, semplici primitive (una primitiva e' una funzione base con una sintassi e una semantica ben definite)
 - ♦ ricercare l'ortogonalita' tra le primitive (uno ed un solo modo per fare una data operazione)
 - ♦ ricercare la consistenza nei parametri, nelle operazioni, etc (vedi funzioni sulle stringhe della libreria standard C definita in string.h)

Come si definisce un'interfaccia - II

- ♦ Quali informazioni debbono rimanere private?
 - ♦ nascondere i dettagli dell'implementazione
 - rimanete liberi di cambiare l'implementazione se necessario (per es., per motivi di prestazioni)
 - ♦ limitare al minimo (zero, ove possibile) le variabili globali
 - possono nascondere gli effetti collaterali di un'invocazione
 - ♦ limitare al minimo (zero, ove possibile), le informazioni di stato (e.g., variabili locali statiche)
 - e' troppo facile perdere traccia dello stato di ogni libreria o modulo che state utilizzando

Come si definisce un'interfaccia - III

- ♦ Quali informazioni non debbono essere modificate?
 - ♦ limitare l'accesso a risorse esterne che non siano parte esplicita del *contratto* (e.g., file nascosti)
 - limitate la possibilita' di effetti collaterali indesiderati ed interferenza con altri moduli o funzioni anche di sistema
 - ♦ evitare di modificare l'input (esempio negativo: **strtok**)
 - prima o poi rischiate di dimenticarvelo
- ♦ Come vanno gestite la memoria e le altre risorse condivise (file, etc.)?
 - ♦ le risorse debbono essere liberate nello stesso livello logico che le ha allocate

La gestione degli errori - I

- Come si gestiscono gli errori?
 - restituendo un valore non permesso (NULL, -1, etc...)
 - oppure utilizzando una variabile globale (e.g., **errno**)
 - bisogna sempre verificare i valori di ritorno dopo l'invocazione dell'interfaccia!
 - evitare di lasciare eccezioni non gestite
- La verifica della correttezza dei parametri ricevuti (*defensive programming*) aiuta anche la sicurezza

La gestione degli errori - II

- ♦ Chi deve gestire gli errori?
 - ♦ gli errori debbono essere rilevati immediatamente, al piu' basso livello possibile...
 - ♦ ... ma debbono essere gestiti al piu' alto livello possibile, fornendo all'utente (ed al programmatore!) tutte le informazioni di contesto:
 - ♦ dove e' avvenuto l'errore (funzioni invocate)
 - ♦ il tipo dell'errore (e.g., memoria insufficiente, file not found, ...)
 - ♦ perche' e' avvenuto (richiesti 100MB, file "/a/b/c/d.e", ...)
 - ♦ eventuali altre informazioni utili sul contesto (stack, altre variabili utili)
 - ♦ esempio:
**markov: can't open psalm.txt in /home/francesco:
No such file or directory**

La scelta della struttura dei dati e degli algoritmi

Usare la struttura dei dati piu' semplice che permetta un'implementazione efficiente delle funzionalita' desiderate (e.g.: ricerca, inserzione, etc.).

KEEP IT SIMPLE

Linee guida per la scelta

- ♦ I passi principali sono:
 - ♦ selezionare algoritmi e strutture dati che risolvano il problema
 - ♦ considerare la quantita' di dati da gestire od elaborare
 - ♦ preferire tecniche e strutture dati semplici
 - ♦ preferire l'utilizzo di librerie gia' disponibili e ben collaudate
 - ♦ solo se necessario, utilizzare strutture o algoritmi di maggiore complessita'
 - ♦ utilizzare un approccio modulare al fine di limitare l'ampiezza degli interventi sul codice (mantenendo invariata l'interfaccia)

Esempio: la ricerca tabellare - I

- ♦ La struttura dati piu' semplice per rappresentare una tabella e' ...
... una tabella (per esempio, un vettore di strutture)
- ♦ La ricerca sequenziale puo' essere sufficientemente veloce se abbiamo pochi elementi e/o la effettuiamo raramente
- ♦ Se le inserzioni o modifiche non sono troppo frequenti, la tabella puo' essere mantenuta ordinata, permettendo cosi' l'uso della ricerca binaria che e' molto efficiente
- ♦ Strutture dati piu' complesse (come le liste linkate o le hash table) vanno utilizzate dopo averne verificato i vantaggi nella nostra applicazione
- ♦ Un approccio modulare permette di variare la struttura dei dati anche dopo la prima implementazione

Esempio: la ricerca tabellare - II

Tabella di parsing HTML: una tabella ordinata (statica) permette la ricerca binaria

```
typedef struct Nameval Nameval
struct Nameval {
    char    *name;
    int     value;
}
/* HTML characters table */
/* e.g. Aelig is ligature of A and E. */
/* Values are Unicode/ISO10646 */
Nameval htmlchars[] = {
    Aelig ,      0x00c6,
    Aacute ,     0x00c1,
    ...
    zeta  ,      0x03b6,
};
```

```
int lookup(char *name, Nameval tab[], int
ntab)
{
    int low, high, mid, cmp;
    low = 0;
    high = ntab - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        cmp = strcmp(name, tab[mid].name);
        if (cmp < 0)
            high = mid - 1;
        else if (cmp > 0)
            low = mid + 1;
        else /* found match */
            return mid;
    }
    return -1 /* no match */
}
```

Esempio: l'ordinamento

- La ricerca binaria richiede un vettore o lista ordinata
- Non (ri)scrivere codice che e' gia' disponibile in libreria:
 - **qsort** - sorts an array
 - **bsearch** - binary search of a sorted array

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

```
void qsort(void base, size_t nmem, size_t size,  
           int(*compar)(const void *, const void *));
```

```
void *bsearch(const void *key, const void *base, size_t  
              nmem, size_t size, int(*compar)(const void *, const  
              void *));
```

Lo stile

Il codice viene sia letto che scritto, perciò esso deve esprimere chiaramente ciò che fa e come lo fa. I commenti serviranno a chiarire perché lo fa.

SEMPLICITA', CHIAREZZA E GENERALITA'

I nomi - I

- ♦ Definire uno standard di nomenclatura:
 - ♦ usare gli stessi criteri per le abbreviazioni e la separazione tra parole che compongono un nome (uso delle maiuscole o di "_"), etc.
 - ♦ se definisco **UserQueue**, per coerenza definiro' i nomi della funzione **ExitQueue**, non **ExitQ** o **Exit_Queue**
- ♦ Qualunque sia lo standard scelto, attenervisi scrupolosamente
- ♦ I nomi delle variabili devono essere sostantivi e aggettivi, i nomi delle funzioni debbono contenere verbi (e.g., **TotalCapacity**, **AddItem**)
- ♦ Non usare una stessa variabile per scopi diversi
- ♦ Non attribuire ad una variabile locale lo stesso nome di una variabile globale

I nomi - II

- ♦ Usare nomi che siano:
 - ♦ espressivi del significato funzionale: esempio negativo **MyVar**
 - ♦ accurati rispetto alla funzione svolta oppure ai dati contenuti
- ♦ Usare nomi descrittivi per le variabili globali (e.g.: **UserQueue**)
- ♦ Usare nomi brevi per le variabili locali (e.g.: **n**, **i**)
- ♦ I nomi delle costanti debbono essere interamente maiuscoli (e.g.: **MAXNUMITEM**)
- ♦ Evitare di inserire nei nomi informazioni non funzionali, per esempio:
 - ♦ informazioni sul tipo della variabile - **MyInt**
 - ♦ informazioni sulla struttura dei dati - **LinkedList2**)

Indentazione e parentesi graffe

- ♦ Indentare per mostrare la struttura
 - ♦ almeno 3-4 caratteri, meglio un tab
 - ♦ usare il corretto livello di indentazione per tutti i blocchi di istruzioni
 - ♦ se avete piu' di 3 livelli di annidamento (e quindi di indentazione), probabilmente il codice e' troppo complesso e deve essere ristrutturato o si devono creare funzioni che raggruppano blocchi di istruzioni
- ♦ Usare in maniera chiara e consistente le parentesi graffe, per es.
 - ♦ { alla fine della linea: **if** (...) { oppure **do** {
 - ♦ } all'inizio della linea, seguito eventualmente solo dalla continuazione della stessa istruzione: } oppure } **while** (...)
 - ♦ eccezione: nella dichiarazione di funzione i caratteri { e } usano una riga intera

Espressioni ed istruzioni - I

- Non scrivere piu' istruzioni sulla stessa linea
- Non scrivere piu' di 80 caratteri per linea
- Usare parentesi e spazi vuoti oculatamente al fine di chiarire ogni possibile ambiguita' di interpretazione
- Le espressioni seguenti sono entrambe corrette, qual'e' piu facile da leggere?

```
leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

```
leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```
- Evitare effetti collaterali (i seguenti esempi non funzionano, perche'?)
 - `str[i++] = str[i++] = ' ';`
 - `scanf( %d %d , &yr, &profit[yr]);`

Espressioni ed istruzioni - II

- ♦ Usare espressioni facilmente leggibili:
 - ♦ limitare l'uso delle negazioni nelle espressioni logiche
 - ♦ spezzare le espressioni complesse in piu' termini
 - ♦ semplificare le espressioni ove possibile
 - ♦ non abusare dell'operatore ?
 - ♦ la lunghezza non e' un criterio di chiarezza (vedi esempi nel lucido seguente)

Espressioni ed istruzioni - III

- Le espressioni seguenti sono entrambe corrette, qual e' piu' facile da leggere?

```
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

- Cosa significa l'espressione seguente?

```
child=( !LC&&!RC)?0:( !LC?RC:LC);
```

Gli idiomi

- Come nei linguaggi naturali, anche nella programmazione un idiomma e' il modo convenzionale di esprimere un concetto
- Ogni linguaggio presenta molti idiomi, che conviene conoscere ed utilizzare in maniera consistente
- L'uso di un idiomma rende agevole la lettura (e quindi il debugging)
- Il mancato uso di un idiomma evidenzia un'anomalia e richiama immediatamente l'attenzione del lettore
- Gli idiomi coprono anche l'invocazione di funzioni (che devono restituire un intero):
 - usare un return code (0 indica successo, -1 indica errore)
 - usare un pattern costante (e.g., prima i parametri in scrittura, poi in lettura)

Esempi di idiomi - I

- L'idioma per un ciclo su di un vettore e':

```
for (i = 0; i < n; i++)  
    array[i] = 1.0;
```

- L'idioma per un ciclo su di una lista e':

```
for (p = list; p != NULL; p++)  
    ...
```

- Gli idiomi per un ciclo infinito (da usare solo nel caso in cui la condizione di uscita sia asincrona e non gestibile) sono:

```
for (;;) 
```

```
while (1)
```

Esempi di idiomi - II

- L'idioma per eseguire un'assegnazione in un ciclo e':

```
while ((c = getchar()) != EOF)  
    putchar(c);
```

- L'idioma per allocare memoria per un stringa e' (a cui aggiungere il controllo sul valore di ritorno della **malloc**):

```
p = malloc(strlen(buf)+1);  
    strcpy(p, buf);
```

- L'idioma per una decisione a molte strade e' **if ... else if ... else**, non una sequenza di **if** annidati
- Un'alternativa e' l'uso di **switch ... case**, dove ogni caso deve essere terminato da **break** (eccetto che per blocchi di istruzioni identici)

Le macro

- ♦ Evitare l'uso di funzioni in forma di macro:
 - ♦ possono forzare valutazioni multiple, con sottili bug ...

```
#define isupper(c) ((c) >= 'A' && (c) <= 'Z')  
...  
while (isupper(c = getchar()))  
    ...
```

- ♦ ... o potenziali problemi di prestazioni

```
#define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?  
0.5:-0.5)))  
...  
size = ROUND_TO_INT(sqrt(dx*dx + dy*dy))
```

I numeri *magici*

- I numeri magici sono le costanti del programma e meritano tutti un nome
- I numeri magici vanno definiti come costanti, non macro
- Il codice sorgente non dovrebbe contenere costanti eccetto (ma solo in alcuni casi) 0 e 1
- Eccezioni:
 - NO: `str = 0; name[i] = 0; x = 0;`
 - SI: `str = NULL; name[i] = '\0'; x = 0.0;`
- Usare il linguaggio (`sizeof`) per calcolare le dimensioni di un oggetto
 - `#define NELEMS(array) (sizeof(array) / sizeof(array[0]))`

I commenti - I

- ♦ I commenti devono aiutare a capire il codice evidenziando punti importanti e fornendo il quadro dell'elaborazione
- ♦ Il commento non deve sostituirsi alla leggibilità del codice, deve integrarla con ulteriori informazioni, quindi ...
- ♦ ... i commenti non devono ripetere ciò che il codice già dice ma aggiungere qualcosa che non è ovvio
- ♦ Ogni oggetto deve essere commentato: funzioni, strutture dati, variabili globali e locali
- ♦ Verificare costantemente che il codice ed i commenti siano allineati e non si contraddicano

I commenti - II

- Ogni funzione deve iniziare con un commento contenente:
 - obiettivo della funzione ed algoritmi o tecniche usate con eventuali riferimenti (manuali, bibliografia, etc)
 - significato di tutti i parametri
 - variabili globali accedute (e se vengono modificate)
 - significato delle variabili locali non banali
 - possibili valori di ritorno e loro significato
- I commenti devono evidenziare la separazione di blocchi diversi di codice
- Ogni linea o blocco di codice non banale merita, di norma, un commento

Esempio di codice commentato

```
#include stdlib.h

/* emalloc: invoke malloc and print message to stderr */
/*           in case of error; syntax is compatible */
/*           with malloc so can be linked instead */
/* global variables: none */
void *emalloc(size_t n)
{
    void p;           /* pointer to allocated memory */
    p = malloc(n);
    if (p == NULL) {
        fprintf(stderr, "malloc of %d bytes failed! ", n);
        return p;
    }
}
```

Debugging e collaudo

Il tempo speso nel debugging raramente e' minore del tempo speso a scrivere codice. Dobbiamo imparare dai nostri errori.

MELIUS PREVENIRE QUAM CURARE

Cosa fare per evitare che qualcosa vada storto...

Diminuire le probabilita' che questo succeda:

- pensare prima di scrivere il codice e leggerlo attentamente prima di modificarlo
- verificare attentamente le piu' comuni fonti di errore (e.g., passaggio per indirizzo in **scanf**, tipi passati a **printf**, ordine dei parametri nell'invocazione, uso di **=** al posto di **==**, uso di variabili non inizializzate, indici di un vettore esterni all'intervallo corretto, ...)
- ogni volta che si trova un errore, verificare che non sia stato commesso uno simile anche altrove
- spiegare il proprio codice a qualcun'altro
- verificare con carta e penna i casi piu' semplici

Cosa fare per scoprire se qualcosa sta andando storto... - I

- ♦ Mentre si scrive il codice di ogni funzione:
 - ♦ verificare pre- e post-condizioni
 - ♦ usare **assert**
 - ♦ programmare in maniera difensiva (verificare i parametri in input, le variabili globali, etc.)
 - ♦ verificare i codici di ritorno delle funzioni invocate
- ♦ Predisporre un ambiente di collaudo (*scaffolding*) in cui verificare il codice:
 - ♦ nei casi limite (file vuoto, lista vuota, inizio o fine del vettore, etc.)
 - ♦ nei casi piu' significativi
 - ♦ nei casi piu' critici (e.g., vicino a valori importanti)

Cosa fare per scoprire se qualcosa sta andando storto... - II

Per l'intera applicazione e per ogni suo modulo:

- predisporre un ambiente di collaudo (automatico!) in cui eseguire un collaudo sistematico
- effettuare verifiche incrementali (ad ogni modifica)
- verificare prima le parti semplici per ottenere rapidamente una struttura funzionante
- verificare proprietà globali o di conservazione (e.g., numero di record scritti)
- se possibile, confrontare due implementazioni indipendenti (e.g., con un prototipo sviluppato in ambiente interpretato)
- verificare che una modifica non abbia introdotto errori (*regression testing*)
- provare input improbabili, errati o enormi (*stress testing*)

Cosa fare se qualcosa e' andato storto...

- Intervenire appena si evidenzia un errore (e.g., non ignorare i crash imprevisti!)
- Rendere l'errore riproducibile
- Iniziare i controlli dalle modifiche piu' recenti
- Utilizzare un metodo di bisezione per trovare la sezione di codice incriminata
- Controllare i *soliti ignoti* (parametri, variabili globali, passaggio per indirizzo, indici out-of-bound etc.)
- Studiare le caratteristiche statistiche dell'errore
- Stampare messaggi di verifica e trace, meglio su un log file tramite opzione
- Inserire controlli sulle variabili in ingresso e sui parametri piu' significativi (lunghezze di liste, etc.)
- Usare il debugger...

Appendice:

**Compendio di regole per la
progettazione e implementazione
del software in ambiente UNIX/Linux**

Basi della filosofia Unix

- Il sistema operativo Unix e le sua varianti hanno il loro punto di forza in una filosofia di base nell'approcciare l'analisi ed il progetto del software
- La filosofia Unix non si basa su un metodo formale di progettazione ma piuttosto si è evoluta con un metodo bottom-up, in cui la prassi e l'esperienza hanno portato alla identificazioni di alcune “regole generali” che è bene tenere sempre a mente
- Tali regole sono utili a qualsiasi programmatore in qualsiasi ambiente di sistema operativo
- Tali regole sono esposte e spiegate nel classico *The Art of UNIX Programming* di Eric S. Raymond, uno degli sviluppatori di Unix dal 1982

La regole per la progettazione ed implementazione del SW sotto UNIX/Linux - I

1. Modularity: Write simple parts connected by clean interfaces.
2. Clarity: Clarity is better than cleverness.
3. Composition: Design programs to be connected to other programs.
4. Separation: Separate policy from mechanism; separate interfaces from engines.
5. Simplicity: Design for simplicity; add complexity only where you must.
6. Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

La regole per la progettazione ed implementazione del SW sotto UNIX/Linux - II

7. Transparency: Design for visibility to make inspection and debugging easier.
8. Robustness: Robustness is the child of transparency and simplicity.
9. Representation: Fold knowledge into data so program logic can be stupid and robust.
10. Least Surprise: In interface design, always do the least surprising thing.
11. Silence: When a program has nothing surprising to say, it should say nothing.
12. Repair: When you must fail, fail noisily and as soon as possible.

Le regole per la progettazione ed implementazione del SW sotto UNIX/Linux - III

13. Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Optimization: Prototype before polishing. Get it working before you optimize it.
16. Diversity: Distrust all claims for "one true way".
17. Extensibility: Design for the future, because it will be here sooner than you think.

Lecture consigliate

Programmazione e progettazione:

- Brian W. Kernighan, Rob Pike
The Practice of Programming
Addison-Wesley
- Niklaus Wirth
Algorithms + data structures = programs
Prentice Hall
- Linus Torvalds
Coding Style
nel *tarball* del kernel di Linux
- Richard Stallman et al.
The GNU Coding Standard
da <http://www.gnu.org>
- Eric Steven Raymond
The Art of Unix Programming
Addison-Wesley

Metodologia e documentazione:

- G. Polya
How to Solve it
Princeton Science Library
- Frederick P. Brooks,
The Mythical Man-Month
Addison-Wesley
- Edward R. Tufte
Visual Explanations
Graphics Press
- AA. VV.
Developing Quality Technical Information
IBM Press