LA “CRISI del SOFTWARE”

Programmi di piccole dimensioni (in the small)

- **enfasi sull’*algoritmo***
- **programmazione strutturata**

LA “CRISI del SOFTWARE”

Programmi di medie dimensioni

- **funzioni e procedure come astrazioni di espressioni/istruzioni complesse**
- **decomposizione degli algoritmi in blocchi funzionali**
- **accoppiamento dati/funzioni:**
  - **Variabili locali**
  - **Passaggio di Parametri**
  - **Variabili Globali**

LIMITI del MODELLO PROCEDURALE

- Il modello procedurale è basato su un dualismo di fondo: decomposizione di un programma in due tipi di entità:
  - strutture dati (entità passive che contengono informazioni)
  - procedure (entità attive che modificano le informazioni)
- Il problema è che queste due entità sono scollegate fra di loro: se dichiariamo una variabile globale, tutte le procedure di un'applicazione possono modificarla senza controllo: manca il concetto di protezione dei dati
- In un'applicazione complessa il fatto che in ogni punto si possa modificare qualunque dato può facilmente generare una situazione incontrollabile

ESEMPIO su 3 FILE

File **`main.c`**

```
float fahrToCelsius(float f);
main() { float c = fahrToCelsius(86); }
```

File **`f2c.c`**

```
extern int trentadue;
float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}
```

File **`32.c`**

```
int trentadue = 32;
```

ESEMPIO

- **un file `dispari.c` che definisca la funzione *e una variabile globale* che ricordi lo stato**
- **un file `dispari.h` che dichiari la funzione**

dispari.c

```
int ultimoValore = 0;

int prossimoDispari(void){
  return 1 + 2 * ultimoValore++; }
```

(sfrutta il fatto che i dispari hanno la forma 2k+1)

dispari.h

```
int prossimoDispari(void);
```


ESEMPIO rivisitato

Realizzazione alternativa del componente:

dispari.c

```
static int ultimoValore = 0;

int prossimoDispari(void){
  return 1 + 2 * ultimoValore++;
}
```

(dispari.h non cambia)

ESEMPIO rivisitato (2)

Realizzazione alternativa del componente:

dispari.c

```
int prossimoDispari(void){
   static int ultimoValore = 0;
 return 1 + 2 * ultimoValore++;
}
```

(dispari.h non cambia)

LA “CRISI del SOFTWARE”

Programmi di grandi dimensioni (in the large)

- **trattano grandi quantità di informazioni**
  - **la decomposizione funzionale è inadeguata**
  - **dati e funzioni che elaborano i dati sono *scorrelati*: nulla indica che le une agiscano sugli altri**
- **devono essere sviluppati da gruppi, ma**
  - **la decomposizione funzionale e il disaccoppiamento dati/funzioni non agevolano la decomposizione del lavoro**

(segue)

LA “CRISI GESTIONALE”

Il costo maggiore nel processo di produzione del software è dovuto alla manutenzione

– correttiva (per eliminare errori)
– adattativa (per rispondere a nuove esigenze)

Programmi di piccole dimensioni

– **trovare gli errori non è difficilissimo**
– **l’impatto delle modifiche è intrinsecamente limitato dalle piccole dimensioni del programma**

LA “CRISI GESTIONALE”

Programmi di medie dimensioni

- **individuare gli errori è già più complesso**
- **l’impatto delle modifiche si propaga, a causa del non-accoppiamento dati/funzioni, anche a funzioni o procedure *diverse* da quella modificata**

Programmi di grandi dimensioni

- **trovare gli errori può essere *estremamente difficile e oneroso***
- **data la propagazione delle modifiche, ogni cambiamento coinvolge *tutto il team di sviluppo***

LA “CRISI GESTIONALE”

Programmi di medie dimensioni

- **individu**
- **l’impatt** **sa del non-ac** **fun-zioni o** **cata.**

È necessario cambiare RADICALMENTE il modo di concepire, progettare e programmare il software

Programmi di grandi dimensioni

- **trovare gli errori può essere *estremamente difficile e oneroso***
- **data la propagazione delle modifiche, ogni cambiamento coinvolge *tutto il team di sviluppo***

OBIETTIVO

Costruzione di software

ben organizzato, modulare, protetto, riusabile, riconfigurabile (dinamicamente?), flessibile, documentato, incrementalmente estendibile, …

L'enfasi non è più tutta / solo / prioritariamente su efficienza e su ottimizzazione

ESEMPIO

- Facciamo un esempio: un contatore
- Vediamo, passo dopo passo, come si possa migliorare l'organizzazione del programma

ESEMPIO: VERSIONE 0

```
main.c
#include <stdio.h>
typedef struct {
  int valore; /* valore attuale */
  int step;
} Contatore;

void main ()
{Contatore C;
 C.valore=vcont;
 C.step=2;
 printf("%d\n", C.valore);
 C.valore+=C.step;
 printf("%d\n", C.valore);
}
```

ESEMPIO: VERSIONE 1

```
main.c
#include <stdio.h>
typedef struct {
  int valore; /* valore
   attuale */
  int step;
} Contatore;

void init(int vcont, int stp);
int  getValue();
void incre();

Contatore C;
void main ()
{
 init(0, 2);
 printf("%d\n", getValue());
 incre();
 printf("%d\n", getValue());
}
```

```
void init (int vcont, int
   stp)
{
 C.valore=vcont;
 C.step=stp;
}
int getValue()
{
 return C.valore;
}
void incre ()
{
 C.valore+=C.step;
}
```

DOMANDE

- E se volessimo cambiare la struttura contatore? Dove viene manipolato? Solo nelle funzioni o in tutto il codice?
- Se devo scrivere un altro programma che ha bisogno del tipo di dato contatore che cosa posso riutilizzare immediatamente del codice?
- Se ho bisogno di un contatore che ha ore, minuti e secondi posso riutilizzare le funzioni scritte?

VERSIONE 1 - CONSIDERAZIONI

- E’ tutto nel main
- Non abbiamo alcuna forma di separazione e di protezione ….nulla mi vieta di fare:

```
main.c
……
void main ()
{
 init(0, 2);
 printf("%d\n", getValue());
 incre();
 printf("%d\n", getValue());
 C.valore+=10;  (ma se volevo un
   incremento sempre predefinito?)
}
```


OLTRE LA PROGRAMMAZIONE STRUTTURATA

MODULARITÀ ===> programmare per parti (MODULI)

Nasce come metodologia di programmazione *"in the large"*: un team di programmatori può suddividere il problema in sottoproblemi e, quindi, ogni programmatore si può concentrare sulla soluzione di un sottoproblema

ESEMPIO – VERSIONE 2/1

contatore.h

```
typedef struct {
               int valore; /*
   valore attuale */
                int step;
                     } Contatore;

Contatore C;

void init (int vcont, int stp);
int  getValue();
void incre ();
```

contatore.c

```
#include "contatore.h"
void init (int vcont, int stp)
{
 C.valore=vcont;
 C.step=stp;
}
int getValue()
{
 return C.valore;
}
void incre ()
{
 C.valore+=C.step;
}
```


ESEMPIO – VERSIONE 2/2

main.c

```
#include <stdio.h>
#include "contatore.h"

void main ()
{
  init(0, 2);
  printf("%d\n", getValue());
  incre();
  printf("%d\n", getValue());
}
```

→

main.c

```
…
typedef struct {
               int valore; /*
   valore attuale */
                int step;
                     } Contatore;
Contatore C;
void init (int vcont, int stp);
int  getValue();
void incre ();

void main ()
{
  init(0, 2);
  printf("%d\n", getValue());
  incre();
  printf("%d\n", getValue());
}
```

VERSIONE 2 - CONSIDERAZIONI

Abbiamo fatto un primo passo:

- lavoriamo con più files...
- contatore globale definito nel .h
- funzioni separate dal main

ESEMPIO – VERSIONE 3/1

contatore.h

```
void init (int vcont, int stp);
int  getValue();
void incre ();
```

contatore.c

```
typedef struct {
int valore;  int step;
} Contatore;
Contatore C;
void init (int vcont, int stp)
{
C.valore=vcont;
C.step=stp;
}
int getValue()
{
return C.valore;
}
void incre ()
{
C.valore+=C.step;
}
```

ESEMPIO – VERSIONE 3/2

main.c

```
#include <stdio.h>
#include "contatore.h"

void main ()
{
 init(0, 2);
 printf("%d\n", getValue());
 incre();
 printf("%d\n", getValue());
}
```


main.c

```
#include <stdio.h>
void init (int vcont, int stp);
int  getValue();
void incre ();

void main ()
{
 init(0, 2);
 printf("%d\n", getValue());
 incre();
 printf("%d\n", getValue());
}
```


VERSIONE 3 - CONSIDERAZIONI

- E’ un tipo di dato astratto
- Abbiamo information hiding
- Ma… possiamo usare solo un contatore

ESEMPIO – VERSIONE 4/1

contatore.h

```
typedef struct {
  int valore;
  int step;
} Contatore;

void init (int vcont, int stp,
   Contatore &Cont);
int  getValue(Contatore Cont);
void incre (Contatore &Cont);
```

contatore.c

```
#include "contatore.h"
void init (int vcont, int stp,
   Contatore *Cont)
{
 (*Cont).valore=vcont;
 (*Cont).step=stp;
}
int getValue(Contatore Cont)
{
 return Cont.valore;
}
void incre (Contatore *Cont)
{
 (*Cont).valore+=(*Cont).step;
}
```


ESEMPIO – VERSIONE 4/2

main.c

```
#include <stdio.h>
#include "contatore.h"
void main ()
{
 Contatore C1, C2;
 init(0, 2, &C1);
 init(0, 1, &C2);
 printf("c1 %d\n", getValue(C1));
 incre(&C1);
 printf("c1 %d\n", getValue(C1));
 printf("c2 %d\n", getValue(C2));
 incre(&C2);
 printf("c2 %d\n", getValue(C2));
 incre(&C2);
 printf("c2 %d\n", getValue(C2));
 incre(&C2);
 printf("c2 %d\n", getValue(C2));
}
```

→

main.c

```
…
typedef struct {
  int valore;
  int step;
} Contatore;

void init (int vcont, int stp,
   Contatore &Cont);
int  getValue(Contatore Cont);
void incre (Contatore &Cont);

void main ()
{
 Contatore C1, C2;
 init(0, 2, &C1);
 init(0, 1, &C2);
 ….
}
```

VERSIONE 4 - CONSIDERAZIONI

- E' quasi un ADT
- Le funzioni sono separate dal main
- E'possibile definire più contatori
- Ma… manca information hiding

ESEMPIO – VERSIONE 5/1

contatore.h

```
typedef struct Contatore *
   ContHandler;
/* contatore non è definito
   qui */

void init (int vcont, int
   stp, ContHandler *Cont);
int  getValue(ContHandler
   Cont);
void incre (ContHandler
   Cont);
```

contatore.c

```
#include "contatore.h"
#include <malloc.h>
#include <stdio.h>

typedef struct {
  int valore; int step;
} Contatore;

void init (int vcont, int stp, ContHandler *Cont)
{
 (* Cont) = (Contatore *) malloc
   (sizeof(Contatore));
 ((Contatore *)(*Cont))->valore=vcont;
 ((Contatore *)(*Cont))->step=stp;
}

int getValue(ContHandler Cont)
{ return ((Contatore *)Cont)->valore; }

void incre (ContHandler Cont)
{ Contatore app;
  app = * (Contatore *)Cont;
  app.valore+=app.step;
  * (Contatore *)Cont = app;
}
```

ESEMPIO – VERSIONE 5/2

main.c

```
#include <stdio.h>
#include "contatore.h"

void main ()
{
 ContHandler C1, C2;
 init(0, 2, &C1);
 init(0, 1, &C2);

 printf("c1 .... %d\n", getValue(C1));
 incre(C1);
 printf("c1 .... %d\n", getValue(C1));

 printf("c2 .... %d\n", getValue(C2));
 incre(C2);
 printf("c2 .... %d\n", getValue(C2));
 incre(C2);
 printf("c2 .... %d\n", getValue(C2));
 incre(C2);
 printf("c2 .... %d\n", getValue(C2));
}
```

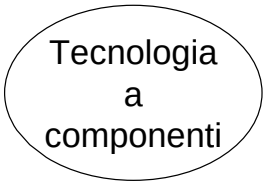

VERSIONE 5 - CONSIDERAZIONI

- Abbiamo quello che volevamo:
  - E' un ADT
  - Rispettiamo l'information hiding
  - Separazione fra funzioni e main
- ... ma è complesso da realizzare...
- ... con il trucco usato dobbiamo gestire una forma di garbage collection...
- ... è meglio lavorare con una metodologia che implementa in modo "naturale" questi concetti.

PRINCIPI STRUTTURALI

- **Modularità**
- **Information hiding, incapsulamento**

- ***Astrazione* vs. *rappresentazione***
- **"*Cosa fa*" vs. "*come lo fa*"**
  - a livello di "dati"
  - a livello di "elaborazione"



INTERFACCIA e IMPLEMENTAZIONE

- **L'*interfaccia* esprime una *vista astratta* di un ente computazionale, nascon-dendo**
  - **organizzazione interna**
  - **dettagli di funzionamento**
- **L'*implementazione* esprime**
  - **la *rappresentazione dello stato interno***
  - **il codice**

  di un ente computazionale

ASTRAZIONE

- **Si focalizza sul *funzionamento osservabile dall'esterno* di un componente identificando gli aspetti piu' significativi**
  - la *struttura interna* di un servitore è *inessenziale* agli occhi del cliente
  - basta assicurare il *rispetto del contratto* stabilito dall'*interfaccia*

INCAPSULAMENTO

- **Lo scopo dell'information hiding è:**
- **Nascondere dettagli implementativi**
- **Proteggere da cambiamenti non controllati**
- **Incapsulamento significa che la *rappresentazione concreta* di un dato può essere modificata *senza che vi siano ripercussioni* sul resto del programma**

Astrazione e incapsulamento

EVOLUZIONE MODELLO PROCEDURALE

- Anche lavorando in C, quindi con il modello procedurale ci si è presto resi conto che era necessario operare in modo modulare
- Nel tempo si sono quindi diffuse pratiche di progettazione e di programmazione basate sui concetti che abbiamo appena esposto
- Un bell'esempio di questo approccio è costituito dalla gestione dei file nella libreria standard del C
- Nella gestione dei file troviamo una serie di concetti estremamente interessanti

I FILE in C: Esempio

```
/* Dichiariamo variabili di tipo puntatore a FILE
   (riferimenti) */
   FILE* f1,f2;
/* Con fopen apriamo i file: il sistema alloca
   risorse, "crea" una struttura per gestire il
   file e ci restituisce un puntatore
   (riferimento) */
   f1 = fopen("pippo.txt","r");
   f2 = fopen("pluto.txt","w");
/* Ottenuti i riferimenti li utilizziamo in tutte
   le operazioni successive */
   fscanf(f1,...);
   fprintf(f2,...);
/* Quando non ci servono più chiudiamo i file e
   il sistema libera le risorse allocate: i file
   "cessano di esistere" */
   fclose(f1);
   fclose(f2);
```

I FILE in C: Astrazione

- Non abbiamo nessuna idea di come sia fatta una struttura dati di tipo FILE
- Abbiamo a disposizione una serie di funzioni (primitive) che ci permettono di operare sul file senza sapere come è fatto
- La struttura dati FILE potrebbe cambiare completamente e il nostro programma non ne risentirebbe minimamente
- C'è una netta separazione fra interfaccia e implementazione
- Non ci occupiamo minimamente dei dettagli implementativi ma ci concentriamo su un'astrazione

I FILE in C: Incapsulamento

- Non possiamo agire direttamente sulla struttura dati
- Abbiamo a disposizione solo un "riferimento" alla struttura e possiamo operare su di esso solo attraverso le funzioni primitive
- Questo garantisce che, se le primitive sono state realizzate correttamente, il suo stato è sempre consistente
- Quindi la struttura dati è protetta
- Si dice che la struttura dati è "incapsulata" e il meccanismo di protezione viene chiamato incapsulamento.

I FILE in C: Dinamicità

- I file vengono gestiti in modo dinamico
- Non hanno un tempo di vita limitato e “automatico” come le variabili locali
- Non esistono per tutta la durata del programma (ES. variabili globali)
- Vengono invece creati con fopen e distrutti con fclose
- Il loro tempo di vita viene quindi gestito da chi li utilizza
- All’atto della “creazione” viene allocata un’area di memoria che viene liberata nel momento della “distruzione”
- Come tutto il resto, anche l’allocazione/deallocazione memoria viene gestita dalle primitive e questo garantisce la consistenza

I FILE in C: Istanze, Stato, Comportamento

- Grazie al meccanismo di creazione/distruzione è possibile lavorare in contemporanea con più file
- E’ sufficiente dichiarare più variabili di tipo riferimento a FILE e invocare fopen() più volte memorizzando il valore di ritorno nelle diverse variabili
- Tutte le altre primitive prevedono come primo parametro un riferimento a FILE, e quindi è possibile operare indipendentemente sui vari file aperti
- Quindi: abbiamo diverse entità con un comportamento comune (determinato dalle primitive) ma con uno stato diverso (ogni variabile lavora su un file diverso, posizionato su una riga diversa ecc.)
- Abbiamo più istanze e l’insieme costituito dal tipo FILE e dalla funzione di creazione fopen() costituisce una “matrice” che permette di creare queste istanze

OLTRE il MODELLO PROCEDURALE

- Tutti questi meccanismi permettono di lavorare in maniera modulare e di costruire applicazioni più robuste e meglio organizzate
- Purtroppo i linguaggi procedurali non mettono a disposizione strumenti che obblighino o anche solo incoraggino a lavorare in questo modo
- Tutto è lasciato alla disciplina e all'esperienza di chi scrive le librerie e le applicazioni
- La separazione di fondo tra strutture dati e procedure rende difficoltoso operare correttamente
- Per gestire la complessità bisogna disporre di linguaggi che supportino i modo naturale uno stile di programmazione corretto
- Bisogna passare dal modello procedurale a quello orientato agli oggetti

Il CONCETTO di OGGETTO

Un oggetto è un'entità dotata di stato e di comportamento

- In pratica un oggetto aggrega in un entità unica e indivisibile una struttura dati (stato) e l'insieme di operazioni che possono essere svolte su di essa (**comportamento**)
- E' quindi un'insieme di variabili e di procedure: le variabili vengono comunemente chiamate campi o attributi dell'oggetto
- Le procedure vengono chiamate metodi
- Sparisce il dualismo di fondo del modello procedurale

INCAPSULAMENTO e ASTRAZIONE

- Lo stato di un oggetto:
  - Non è accessibile all'esterno
  - Può essere visto e manipolato solo attraverso i metodi
- Quindi: lo stato di un oggetto è protetto
  → Il modello ad oggetti supporta in modo naturale l'incapsulamento
- Dal momento che dall'esterno possiamo vedere solo i metodi c'è una separazione netta tra cosa l'oggetto è in grado di fare e come lo fa
- Abbiamo quindi una separazione netta fra interfaccia e implementazione
  → Il modello ad oggetti supporta in modo naturale l'astrazione

CLASSI e OGGETTI - 1

- Per poter operare con gli oggetti abbiamo bisogno di un meccanismo che ci consenta di:
  - Definire una struttura e un comportamento
  - Creare un numero qualunque di oggetti con identica struttura e comportamento ma con identità diversa
- Nella programmazione procedurale abbiamo il concetto di tipo: una volta definito un tipo possiamo dichiarare più variabili identiche fra di loro
- Se vogliamo avere un comportamento dinamico ci serve anche un meccanismo di creazione, che si comporti come fopen() per i file in C
- Nell'OOP questi due ruoli vengono svolti da un'entità chiamata classe

CLASSI e OGGETTI - 2

- Quindi una classe è un entità che permette di
  - Definire la struttura e il comportamento di un oggetto
  - Creare un numero qualunque di oggetti con la struttura specificata
- Gli oggetti creati da una classe si chiamano istanze della classe

  →Ogni oggetto è istanza di una qualche classe
- Tutte le istanze di una classe hanno la stessa struttura (lo stato è fatto nello stesso modo) e lo stesso comportamento (l'insieme dei metodi definiti dalla classe)
- Ogni istanza possiede un proprio stato ("contenuto delle variabili") e una propria identità.

CLASSI e OGGETTI - 3

Nel modello "classico" dell'OOP
esistono solo classi e **oggetti**

- Gli oggetti sono entità dinamiche:
  - Possono essere creati in qualunque momento a partire da una classe
  - Vengono distrutti quando non servono più
- Le classi sono invece entità statiche:
  - Sono sempre disponibili durante l'esecuzione di un'applicazione
- Quando si crea un oggetto si ottiene un riferimento all'istanza appena creata
- I riferimenti rappresentano l'unico modo per comunicare con gli oggetti

RAPPRESENTAZIONE delle CLASSI

- In UML le classi vengono rappresentate con un rettangolo
- Nella forma sintetica compare solo il nome della classe mentre nella forma estesa vengono indicati anche i campi e i metodi