Lezione 14

Ingegneria del codice

Ingegneria del codice

- Insieme di metodologie e pratiche per la produzione di codice di qualità
  - Leggibilità
  - Robustezza
  - Manutenibilità

Ingegneria del codice

- In questi lucidi vedremo solo qualche principio, applicandolo al seguente semplicissimo esempio di codice

```
float f(float j, int s,
float a){return s?a*j:a*j/2;}
```

- Cosa calcola la funzione?

Codice di cattiva qualità

- Per quanto sia breve la funzione, non è immediato capirlo
- Come mai?
- A causa della pessima qualità del codice
- Proveremo ad applicarvi i principi di ingegneria del codice che stiamo per introdurre

Formattazione 1/2

- Il primo semplice principio da applicare sempre è che il codice deve essere **formattato in modo opportuno**
- Abbiamo già visto delle semplici regole di indentazione
- Esistono in generale vari stili di formattazione, ciò che conta è sceglierne uno ed applicare sempre quello
- Proviamo quindi a riformattare il codice andando a capo in modo opportuno, indentando ed aggiungendo qualche spaziatura

Formattazione 2/2

```
float f(float j, int s, float a)
{
    return s ? a*j : a*j/2 ;
}
```

- E' già più facile da leggere
- Ma ancora non è chiaro
- Il prossimo principio fa fare un passo cruciale verso la leggibilità

Nomi significativi variabili

- Bisogna usare nomi significativi per le variabili e le costanti con nome
- Applichiamolo al nostro esempio:

```
float f(float altezza, int tipo, float base)
{
     return tipo ?
                 base * altezza
                 :
                 base * altezza / 2 ;
}
```

- Adesso abbiamo probabilmente capito
- Miglioriamo attraverso il prossimo principio

Nomi significativi funzioni

- Bisogna usare nomi significativi per le funzioni
- In particolare, uno degli approcci migliori per i nomi delle funzioni è utilizzare i verbi
  - Ossia definire nomi delle funzioni del tipo *compi_una_data_azione*

Esempio

- Applichiamo di nuovo questo principio al nostro esempio:

```
float calcola_area_rett_triang(float altezza,
                               int tipo,
                               float base)
{
     return tipo ?
                 base * altezza
                 :
                 base * altezza / 2 ;
}
```

Raggruppare concetti

- Cercare sempre di raggruppare concetti correlati anziché disperderli nel programma
  - Ordinare poi gli elementi in ogni gruppo nel modo più opportuno
- Applicato al nostro esempio:

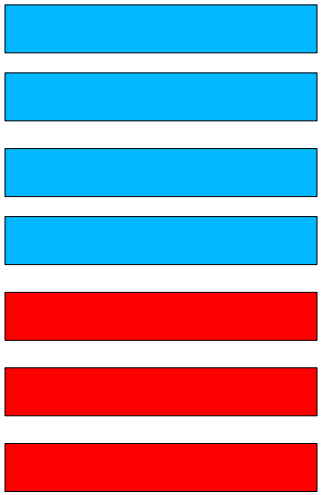
```
float calcola_area_rett_triang(float base,
                               float altezza,
                               int tipo)
{
     return tipo ?
                base * altezza
                :
                base * altezza / 2 ;
}
```

Separare gruppi

- Conviene separare in qualche modo ogni insieme di elementi correlati dall'altro
- Ad esempio, supponiamo che una funzione sia costituita da diverse sequenze di istruzioni, ove ciascuna sequenza è logicamente correlata
- Se alcune di tali sequenze sono contigue, può essere conveniente separarle con una o più righe vuote
- E' invece dannoso inserire righe vuote in contraddizione con la separazione concettuale di sequenze di istruzioni

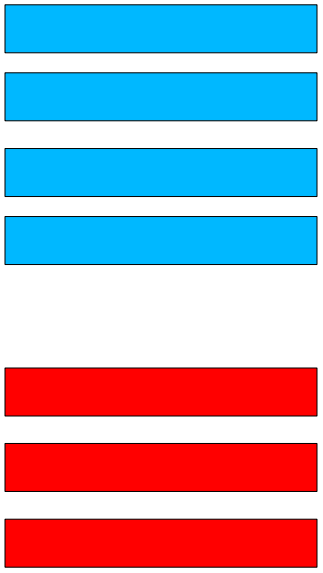
Esempio 1/3

- Supponiamo che ciascuna delle seguenti barre rappresenti una istruzione e che barre dello stesso colore appartengano ad una sequenza logicamente correlata



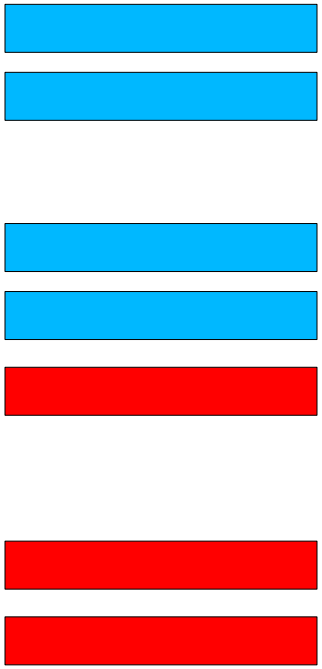
Esempio 2/3

- Ha senso inserire una o più righe vuote per separare le sequenze:



Esempio 3/3

- La seguente è invece una **pessima formattazione**:



Non duplicare il codice

- Bisogna cercare di **non replicare mai** uno stesso frammento di codice in più punti di un programma
  - Anche se ogni duplicato ha delle minime varianti rispetto agli altri
- Infatti, durante la vita del programma il codice duplicato va collaudato e manutenuto, e quindi controllato e modificato, in parallelo
  - Seria fonte di errori
- Soluzione tipica: uso delle funzioni
  - Se si utilizzano nomi appropriati, l'aggiunta di una funzione documenta anche meglio il frammento di codice stesso

Eccezione

- Lasciare pure il codice duplicato se per evitare la duplicazione è necessario rendere il programma ancora più complicato di quanto sarebbe con la duplicazione

Livello di astrazione

- Quale criterio seguire nella scelta di nomi opportuni per gli oggetti in un programma?
- Bisogna utilizzare **nomi appartenenti al dominio del problema**
- Non devono quindi appartenere al dominio dell'implementazione, ossia ad un dominio meno astratto di quello del problema
- Ma neanche ad un dominio più astratto di quello del problema
- Vediamo gli effetti di livelli di astrazione errati nel nostro esempio

Livello troppo basso

- Livello troppo basso:

```
float calcola_prodotto_dimezza(float reale1,
                               float reale2,
                               int non_dimezz)
{
     return non_dimezz ?
                 reale1 * reale2
                 :
                 reale1 * reale2 / 2 ;
}
```

Livello troppo alto

- Livello troppo alto:

```
float calcola_area_figura(float segmento1,
                          float segmento2,
                          int tipo_figura)
{
     return tipo ?
                segmento1 * segmento2
                :
                segmento1 * segmento2 / 2 ;
}
```

Tipo di dato appropriato

- Utilizzare sempre il tipo di dato più appropriato
- Nel nostro esempio, l'uso del tipo di dato **`int`** per il parametro formale **`tipo`** ha contribuito alla poca leggibilità della funzione
  - Quale sarebbe un tipo di dato più appropriato?
  - Un primo passo avanti sarebbe l'uso di un booleano
  - Ma si può fare meglio per migliorare la leggibilità

Esempio

- Si può utilizzare un enumerato

```
enum tipo_figura {rettangolo, triangolo} ;

float calcola_area_rett_triang(float base,
                               float altezza,
                               tipo_figura tipo)
{
     return tipo == rettangolo ?
                 base * altezza
                 :
                 base * altezza / 2 ;
}
```

- A questo punto è praticamente immediato capire cosa fa la funzione

Confronto

- Proviamo a confrontare con la versione iniziale

```
enum tipo_figura {rettangolo, triangolo} ;

float calcola_area_rett_triang(float base,
                               float altezza,
                               tipo_figura tipo)
{
     return tipo == rettangolo ?
                 base * altezza
                 :
                 base * altezza / 2 ;
}
```

```
float f(float j, int s,
float a){return s?a*j:a*j/2;}
```

Nota sul tipo enumerato

- L'uso del tipo enumerato ha anche l'importante vantaggio che, se si aggiunge un enumeratore, non è necessario toccare minimamente tutte le funzioni che lavoravano solo con i precedenti enumeratori
- Nel nostro esempio possiamo scrivere

```
enum tipo_figura {rettangolo, triangolo,
                  cerchio} ;
```

  al posto di

```
enum tipo_figura {rettangolo, triangolo} ;
```

  e la funzione **`calcola_area_rett_triang`** continua ad essere corretta senza bisogno di alcuna modifica (a meno di eventuali controlli di errore che avremmo dovuto mettere da subito)

Nota sul tipo reale

- L'uso del tipo *float* al posto del *double* è giustificato praticamente solo nel caso in cui occupare meno memoria o andare un po' più veloce sia un obiettivo importante
- In tutti gli altri casi è meglio avere una precisione maggiore

Complessità 1/2

- Il termine *complessità* è spesso associato a due diversi significati
- Il primo è quello di **complessità computazionale**
  - Misura il costo di un algoritmo in termini di numeri di passi che deve compiere per ottenere i propri obiettivi
  - Sarebbe più appropriato utilizzare il termine costo computazionale
  - La qualità di un programma è certamente legata al costo computazione degli algoritmi che implementa

Complessità 2/2

- L'altro significato con cui si utilizza il termine complessità riferito ad un frammento di codice, è “quanto è difficile comprendere tale frammento di codice”
  - Tipicamente tale complessità è considerata proporzionale al numero di oggetti che si devono tenere contemporaneamente in mente per comprendere il frammento di codice
- In quanto segue considereremo solo questo secondo significato del termine complessità

Complessità e funzioni

- La complessità è il più grande nemico di ogni progetto software
  - Comporta difficoltà di comprensione, che a sua volta sono fonte di errori
- A questo punto si può capire il ruolo fondamentale delle funzioni, che permettono di
  - spezzare un programma in più componenti
  - poter scrivere ciascun componente senza dover tenere in mente come sono fatti dentro gli altri componenti, ma solo come si usano
- Abbiamo ad esempio usato la funzione *sqrt* o l'operatore << senza bisogno di tenere in mente nessuno dei loro dettagli interni!

Effetti collaterali

- Ecco inoltre perché gli effetti collaterali possono essere così dannosi
  - Non possiamo più spezzare mentalmente il problema!
- Per esempio, se la funzione *sqrt* avesse avuto effetti collaterali, avremmo dovuto stare attenti a tutte le variabili del programma di cui poteva cambiare implicitamente il valore
- In conclusione, per limitare gli effetti collaterali, minimizziamo i passaggi per riferimento senza il qualificatore *const* e l'uso delle variabili globali

Istruzioni di controllo

- Oltre alle variabili globali, la complessità di un frammento di codice è proporzionale al numero di *punti di scelta* presenti
  - Ogni istruzione condizionale o iterativa, in generale una istruzione di controllo, comporta un punto di scelta
- Ecco anche perché istruzioni di controllo molto nidificate sono difficili da leggere
  - Bisogna quindi evitarle
  - Cercare di non superare mai 3 istruzioni di controllo nidificate

Esempi 1/2

```
if (...) {
      ...
      for (...) {
      ...
      }
      ...
}
```

```
if (...) {
      ...
      for (...) {
      ...
            if (...) {
            ...
            }
      }
      ...
}
```

- Esempi di due o tre livelli di nidificazione
  - Sono accettabili

Esempi 2/2

```
if (...) {
        ...
        for (...) {
                ...
                if (...) {
                        ...
                        while(...) {
                                ...
                        }
                        ...
                }
        }
        ...
}
```

- Esempio di quattro livelli di nidificazione
  - Meglio evitare
- Andare ancora oltre è assolutamente da evitare

Riduzione nidificazione

- Una semplice soluzione per evitare o eliminare istruzioni di controllo troppo nidificate è prendere una o alcune delle istruzioni di controllo più interne e spostarle in una funzione
  - Si potranno quindi sostituire tali istruzioni di controllo interne con una invocazione di funzione
  - Notare come le funzioni aiutino di nuovo a dominare la complessità
- In generale può valere la pena di riprogettare il frammento di codice e/o l'algoritmo utilizzato
  - Per esempio, dati due cicli nidificati, spezzare quello interno può aiutare a riscrivere il tutto come due cicli in sequenza ma non nidificati

Prevenzione nidificazione

- Vi sono poi dei semplici accorgimenti per prevenire la nidificazione eccessiva

1) Limitare il numero di colonne occupate dal programma
   - Questo ci spinge a limitare il livello di nidificazione, perché diventa scomodo indentare le istruzioni troppo a destra
   - Utilizzare 80 colonne se si vuole rispettare una delle convenzioni più diffuse

1) Cercare di limitare ad una pagina del proprio editor il numero massimo di righe occupate da un ciclo

Uso dei commenti

- Commentare sempre le parti di codice che necessitano di commenti e solo quelle
  - I commenti devono essere sintetici ma possibilmente completi
    - Troppi commenti sono peggio di nessun commento
  - Un commento non deve ripetere cosa fa il codice, perché per quello basta il codice stesso
    - Un commento deve rendere chiaro **lo scopo** di un frammento di codice
    - O in alternativa può fornire un **riepilogo**

Commenti delle funzioni

- Nel commento di una funzione sarebbe bene non far mancare le seguenti parti
  - Descrizione dello scopo della funzione
  - Descrizione dei parametri di ingresso
  - Descrizione dei parametri di uscita
  - Se non ovvio, descrizione del valore di uscita
  - Descrizione degli eventuali effetti collaterali
- All'aumentare della complessità della funzione può aver senso descrivere il modo in cui ottiene lo scopo per cui è stata definita
- Proviamo ad applicare questi principi al nostro esempio ...

Esempio

```
// tipi possibili di figure geometriche
enum tipo_figura {rettangolo, triangolo} ;

/*
 * Calcola l'area di un rettangolo o di un
 * triangolo.
 * Prende in ingresso la base e l'altezza della
 * figura, nonché il tipo della figura stessa.
*/
float calcola_area_rett_triang(float base,
                               float altezza,
                               tipo_figura tipo)
{
     return tipo == rettangolo ?
          base * altezza  : base * altezza / 2 ;
}
```

Livello di warning 1/2

- Ci sono molti casi in cui si effettuano operazioni 'dubbie' o in qualche modo rischiose in un programma
- Per esempio tutti i casi in cui ci si effettuano operazioni con possibile perdita di informazione, o in cui si utilizza il valore di una variabile senza prima averla inizializzata
- Si può configurare il compilatore per il più alto livello di *warning*, in cui ci segnala quindi ogni possibile *warning* di cui è a conoscenza

Livello di warning 2/2

- Un buon programmatore configura **sempre** il compilatore per il **massimo livello di warning**
- E corregge il programma finché il compilatore, pur configurato in questo modo, non segnala più neanche una warning
- Per attivare il massimo livello di warning col gcc (g++) basta aggiungere l'opzione **`-Wall`**

Organizzazione dati

- Vedremo infine delle regole per organizzare bene le proprie strutture dati nella lezione sul tipo **`struct`**