Organizzazione di un programma.

I buoni programmi sono organizzati in sezioni ognuna delle quali è memorizzata su un file separato, per seguire l'approccio OO.

Ogni classe è definita su due files:

1. un file header per la speficazione, che avrà **.h** come estensione e che conterrà il nome della classe e la specificazione formale dei metodi (i prototipi delle funzioni)

2. un file con l'implementazione, che avrà estensione **.c** e che conterrà gli attributi della classe, nonché il codice dei metodi (funzioni)

Poichè in C possiamo compilare separatamente i files, la specificazione e l'implementazione di una classe possono essere scritte e compilate separatamente da un programma utente della classe. La compilazione separata ha molti vantaggi:

1. poiché compiliamo un file alla volta localizziamo più velocemente gli errori segnalati dal compilatore.
2. Una volta che un file è stato compilato correttamente può essere solamente linkato con altri file oggetto, compilati in un secondo momento, riducendo i tempi complessivi di compilazione.
3. Mettendo a disosizione il file oggetto e non il sorgente proteggiamo il nostro file da modifiche non autorizzate.

Quindi per produrre un programma sfruttando la compilazione separata, dobbiamo:

1. Usare un editor per produrre i files
2. Usare un compilatore per ottenere codice in linguaggio macchina, i files oggetto, che hanno tipicamente un'estensione .o o .obj e che generalmente non sono eseguibili senza risolvere alcuni riferimenti tra i diversi files
3. Usare un linker per ottenere un file eseguibile.

Sotto Unix o Linux

cc -c mio.c

(cc è il nome del compilatore, **C c**ompiler)
-c è un opzione che dice al compilatore di fermarsi dopo aver prodotto il codice oggetto corrispondente al codice C nel file di testo mio.c
Se tutto è andato bene il compilatore ha prodotto un file mio.o.

Per combinare i file oggetto mio.o, tuo.o, suo.o semplicemente si chiama di nuovo il compilatore

cc -o finale mio.o tuo.o suo.o

-o annuncia il nome del file eseguibile

Per esempio per la nostra classe Rettangolo e si dovrebbe fare:
cc -c rettangolo.c
cc -c prova.c (è il nome di un file che contiene un main in cui si usano le funzioni di rettangolo)

cc -o prova prova.o rettangolo.o
(è consuetudine chiamare l'eseguibile con il nome del file che contiene il main)

infine per eseguire il programma si digita semplicemente
prova

Precondizioni e Postcondizioni.

Esponendo l'esempio della classe rettangolo avevamo notato che alcune funzioni non erano completamente specificate fornendo solo il prototipo. Vogliamo quindi introdurre un metodo standard per completare le specificazione dei metodi aggiungendo al prototipo informazioni precise circa i valori di definizione delle funzioni e l'output prodotto. Introduciamo cioè le **precondizioni** e le **postcondizioni**: le prime stabiliscono per quali valori eventualmente la funzione NON è definita e le seconde i valori che

funzione restituisce.

Le possibili specificazioni complete della funzione alt, che restituisce l'altezza del rettandolo, sono

1.

```
double alt(RettangoloP r)
/*prec: r !=NULL
postc: restituisce l'altezza di r */
```

2.

```
 double alt(RettangoloP r)
/*prec: nessuna
postc: se r!=NULL, restituisce l'altezza di r, altrimenti 0.*/
```

Nel primo caso il controllo sul valore NULL del parametro prima di utilizzare la funzione è lasciato alla responsabilità del chiamante, con il vantaggio di avere una funzione alt più semplice.

Nel secondo caso non c'è rischio di segmentatio fault dovuto al valore NULL del parametro in ingresso, ma poi il chiamante dovrebbe gestire il caso del valore 0, restituito dalla funzione.

Quindi qui è preferibile la prima soluzione, ma non è la soluzione ideale in tutti i contesti.

Nel caso del costruttore, le precondizioni sono dettate dal problema:

```
RettangoloP CostRettangolo( double alt, double larg );
/* Prec: alt>0 && larg >0
Postc: restituisce un puntatore al rettangolo di altezza alt
e larghezza larg*/
```

Per esempio nel caso di una funzione che cerca un elemento in una lista potrei avere le seguenti possibilità di scelta :

1.

```
int cerca(ListPtr L, Elem el)
/* verifica se l'elemento el è presente almeno una volta nella lista
*prec: nessuna
*postc:  restituisce 1 se l'elemento è presente nella lista,  0
```

altrimenti *

2.

```
int cerca(ListPtr L, Elem el)
/*  verifica se l'elemento el è presente almeno una volta nella lista
*prec: L != NULL
*postc:  restituisce 1 se l'elemento occorre nella lista,  0 altrimenti
*
```

3.

```
int cerca(Listptr L, Elem el)
/* verifica se l'elemento el è presente almeno una volta nella lista
*postc: se L = NULL restituisce -1, altrimenti
se  l'elemento e' presente restituisce 1, in caso contrario 0.*/
```

Nel primo caso 1, se si ottiene il valore zero non si può sapere se questo era dovuto all'assenza dell'elemento da un insieme non vuoto o da una chiamata della funzione su una collezione vuota, a meno che non sia stato fatto un controllo preliminare alla chiamata. Questa specificazione porta a un'implementazione più efficiente.
Nel secondo caso il controllo preliminare alla chiamata è obbligato dal fatto che altrimenti si rischia un "segmentation fault".
Nel terzo non si corre questo rischio e si ha la possibilità di eseguire un controllo sul risultato per distinguere i due casi.
Quale soluzione è la migliore? Purtroppo non c'è una soluzione che va bene in tutti i casi. Infatti le precondizioni spesso sono introdotte per evitare il controllo di condizioni troppo onerose da verificare. Ovviamente una precondizione non banale rende la funzione parziale e questo può essere un inconveniente per il chiamante che deve assicurarsi di non chiamare la funzione su valori che non soddisfano le precondizioni.
La decisione di utilizzare le precondizioni per gestire casi particolari di dati in input deve essere determinata

- dall'ambito di applicazione della funzione. Se si prevede un uso solo locale di un'applicazione, si può pensare di verificare al momento della chiamata che le precondizioni siano rispettate, se la funzione ha un uso più ampio, bisogna fare un uso molto prudente delle precondizioni.

- dal costo dei controlli. Esempio: la ricerca binaria opera correttamente solo su elementi ordinati, ma se controlliamo questo al momento della chiamata della ricerca binaria perdiamo tutto il vantaggio in termini di prestazioni della ricerca binaria!

Nota che nel caso della specificazione delle funzione cerca abbiamo premesso alle pre e postcondizioni un breve commento che descrive il comportamento della funzione a grandi linee.

Molto spesso è utile aggiungere questi commenti in modo che solo se la funzione è interessante si leggono i dettagli della sua specificazione, attraverso pre e post.