Costruzione di una classe passo dopo passo.

Stabiliamo qui con più precisione i passi da seguire per definire una classe:

1. Specificazione
1.1. Nome della classe (produrre il nome del tipo di dato)
1.2. Specificazione dei metodi (si tratta di fornire i prototipi con le precondizioni e le postcondizioni)
   1.2.1. specificazione prima di tutto di costruttori e distruttori
   1.2.2. specificazione degli altri metodi

2. Revisione e controllo della specificazione.

3. Implementazione
3.1. bisogna importare le librerie standard, quelle personali e gli header file della specificazione
3.2. poi specificare gli attributi della classe (definire la struttura)
3.3. infine definire (codificare) i metodi, senza dimenticare di introdurre un assert per ogni precondizione

Rivediamo il processo di produzione di una classe, illustrandolo con un esempio, una classe per la gestione di taniche.
Passo 1: la specificazione.
La prima cosa da fare consiste nel tradurre formalmente dei requisiti formulati in linguaggio naturale.

Descrizione informale dei requisiti:

1. le taniche devono contenere dei liquidi
2. Le taniche sono di forma rettangolare, quindi sono caratterizzate da altezza, larghezza e profondità. Le dimensioni devono essere fornite per costruire una tanica.
3. Le taniche sono di plastica, le dimensioni di una tanica non possono essere modificate dopo che una tanica è stata costruita
4. Si deve assegnare ad ogni tanica la sua capacità. Questa deve essere inferiore al suo volume, per ragioni di sicurezza.
5. Per aggiungere liquido ad una tanica bisogna che la sua capacità non sia stata raggiunta e non bisogna in ogni caso superarla.
6. Per ogni tanica bisogna poter sapere quanto liquido contiene e quanto liquido può ancora contenere

7. da una tanica si può prelevare una quantità di liquido inferiore o uguale al suo contenuto.

Cominciamo quindi a scrivere il file della specificazione formale. Non definiamo alcuna funzione, ci limitiamo a progettare la classe, creandone una specificazione formale, in C. La prima decisione da prendere è il nome per la classe. Se decidiamo per tanica, allora conviene chiamare il file header che contiene la specificazione tanica.h.

```
/**** Nome del file: Tanica.h ****/

/**** Nome della classe****/
typedef struct tanica * TanicaP;
```

Introdurre questo puntatore ci consente di rinviare il momento della decisione definitiva sugli attributi della classe.

```
/**** Metodi  della classe****/
```

Per ogni metodo si deve

1. decidere cosa restituisce
2. scegliere opportunamente il nome
3. stabilire quali parametri di input servono
4. precisare le precondizioni e
5. le postcondizioni

```
/* Costruttori */

TanicaP CostTan ( double alt, double larg, double prof );
/* restituisce una tanica avente le dimensioni specificate
prec: alt >0 && larg >0 && prof>0
postc: restituisce un puntatore a una tanica, NULL se non c'è spazio di
memoria per una nuova tanica*/

/* Costruttore alternativo, con le stesse funzioni tranne CapMax*/
TanicaP CostTanica (double alt, double larg, double prof,double capMax)
/* restituisce una tanica avente le dimensioni e la capacità massima
specificate
prec: alt >0 && larg >0 && prof>0 && capMax < alt*larg*prof
postc: restituisce un puntatore a una tanica,
NULL se non c'è spazio di memoria per una nuova tanica*/
{TanicaP t;
assert(alt>0);
assert(larg >0);
```

```
assert(prof >0);
assert(capMax < alt*prof*larg);
t = malloc( sizeof(struct tanica) );
if ( t == NULL ) printf("CostTan: memoria insufficiente \n");
        else
        { t->alt = alt;
          t->larg = larg;
          t->prof = prof;
          t->cap_max = capMax;
          t->quant_corr = 0.0;
         }
        return t;
}

/* Distruttori */

void DistrTan (TanicaP t );
/*libera lo spazio di memoria impegnato dal puntatore r*/


/* I metodi che restituiscono le dimensioni di una tanica */
double Alt(TanicaP t );
/*prec: t!= NULL
postc: restituisce l'altezza*/

double Larg(TanicaP t );
/*prec: t!= NULL
postc: restituisce la larghezza*/

double Prof(TanicaP t );
/*prec: t!= NULL
postc: restituisce la profondità*/


double LiqCorr(TanicaP t );
/*prec: t!= NULL
postc: restituisce la quantità di liquido contenuta*/

double DispLiq( TanicaP t );
/*prec: t!= NULL
postc: Restituisce  la quantità di liquido che si può aggiungere nella tanica
*/

double Vol(TanicaP t);
```

```
/*prec: t!= NULL
postc: Restituisce  il volume della tanica */

int CapMax( TanicaP t, double cap );
/* Stabilisce la massima capacità
prec: t != NULL && cap >0
postc:  Se cap < volume della tanica restituisce vero, falso altrimenti
*/


int AggLiq( TanicaP t, double tot );
/* aggiunge liquido a una tanica, se la disponibilità lo consente
prec: : t != NULL && tot >0
postc: restituisce vero se  tot è stato aggiunto alla tanica, falso altrimenti.
*/

int PrelLiq( TanicaP t, double tot );
/*  preleva tot di liquido da una tanica, se tot è minore del liquido contenuto
prec: t != NULL && tot>0
postc: restituisce vero se  tot di liquido può essere prelevato, falso
altrimenti*/
```

Passo 2: controllo.

Ora si controlla se tutte le funzionalità richieste nei requisiti sono presenti e se le precondizioni e le postcondizioni riflettono i vincoli imposti dai requisiti utente, nonchè quelli indotti dalla formalizzazione. Un metodo ovvio consiste nello scorrere i requisiti utente e controllare se sono rispecchiati nella specificazione. Bisogna fare molta attenzione in questa fase, in ogni classe si dovranno introdurre dei metodi per ottenere e per aggiornare gli attributi. Spesso, anche se non è il caso in questo esempio gli aggiornamenti non sono indipendenti, quindi il cambiamento di un valore può comportare quello di un altro. Quindi bisogna controllare di aver definito correttamente i metodi.
Una volta SICURI di aver fatto le scelte giuste passiamo all'implementazione. Diamo al file il nome della classe, con estensione .c.
Il primo passo riguarda tutte le direttive di importazioni di file header, quelli standard e poi quelli particolari della nostra classe. Quest'ultima importazione è particolarmente importante perché così il compilatore è in grado di segnalare eventuali inconsistenze con la specificazione.

Passo 3: implementazione.

```
/*Nome del file: Tanica.c */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Tanica.h"
```

Poiché ci sono delle funzioni booleane definiamo due costanti simboliche TRUE e FALSE, per facilitare la lettura del codice.

```
#define TRUE 1
#define FALSE 0
```

Introduciamo le definizione degli attributi della classe attraverso la definizione di una opportuna struttura. Per le dimensioni delle taniche scegliamo un tipo numerico che dia la massima liberta, il double; poiché le taniche dovranno contenere liquidi anche per gli attributi relativi alla capacità massima e alla quantità di liquido correntemente contenuto nella tanica scegliamo il tipo double.

```
struct tanica {
        double alt, larg, prof;
        double cap_max;
        double quant_corr;
        };
```

Ora si devono definire i metodi, cioè le funzioni, cominciando con i costruttori e i distruttori, poi quelli che estraggono gli attributi e infine quelli che implementano le funzionalità. Non si deve dimenticare di trasformare ogni precondizione in un' invocazione di assert.

In ogni costruttore si dovranno allocare le risorse e provvedere alla inizializzazione di un oggetto della classe. Tipicamente la risorsa è lo spazio di memoria per un elemento del tipo struttura introdotto, ma potrebbe anche far riferimento a files, dispositivi di I/O,…

La specificazione informale dovrebbe suggerire quali valori utilizzare per inizializzare la struttura. L'inizializzazione può essere di due tipi:

- attraverso i parametri del costruttore oppure
- utilizzando dei valori di default.

Nel nostro esempio le dimensioni di una tanica sono fornite in input, mentre capacità massima e quantità di liquido contenuto sono inizializzate a zero, come la ragionevolezza suggerisce. E' essenziale inizializzare TUTTI i

campi della struttura, cioè dare un valore iniziale a tutti gli attributi degli oggetti della classe.

```
/* Costruttore */
TanicaP CostTan (double alt, double larg, double prof)
   {TanicaP t;
   assert(alt>0);
   assert(larg >0);
   assert(prof >0);
   t = malloc( sizeof(struct tanica) );
   if ( t == NULL ) printf("CostTan: memoria insufficiente \n");
   else
        {t->alt = alt;
         t->larg = larg;
         t->prof = prof;
         t->cap_max = t->quant_corr = 0.0;
        }
        return t;
   }
```

Anche se nel presente esempio non è necessario, non è detto che ci si debba limitare a un solo costruttore, potrebbero essere utili anche solo dei cotruttori che creano una copia di un oggetto già esistente o dei costruttori con dei particolari valori per gli attributi.

```
   /* Distruttore */
   void DistrTan( TanicaP t )
        {free( t );}
```

Gli estrattori di attributi sono quei metodi, spesso molto semplici, che restituiscono il valore di un attributo di un oggetto della classe creato con il/un costruttore. Essi sono importanti perché il loro uso consente di nascondere dettagli implementativi all'utente della classe, facilitando eventuali modifiche.

```
double Alt(TanicaP t )
/*prec: t!= NULL
postc: restituisce l'altezza*/
   {assert(t);
    return t->alt;
```

```
   }

double Larg(TanicaP t )
/*prec: t!= NULL
postc: restituisce la larghezza*/
   {assert(t);
    return t->larg;
   }


double Prof(TanicaP t )
/*prec: t!= NULL
postc: restituisce la profondità*/
   {assert(t);
    return t->prof;
   }


double LiqCorr(TanicaP t )
/*prec: t!= NULL
postc: restituisce la quantità di liquido contenuta*/
   {assert(t);
   return t->quant_corr;
   }

double DispLiq( TanicaP t )
/*prec: t!= NULL
   postc: Restituisce  la quantità di liquido che si può aggiungere nella
tanica */
   {assert(t);
    return t->cap_max -   t->quant_corr;
   }
```

I metodi che implementano le funzionalità richieste devono essere implementati in modo da rispettare i vincoli presenti nei requisiti.

```
double Vol(TanicaP t)
/*prec: t!= NULL
postc: Restituisce  il volume della tanica */
{assert(t);
return t->alt*t->larg*t->prof;
   }
```

```
int CapMax( TanicaP t, double cap )
/* Stabilisce la massima capacità
prec: t != NULL && cap >0
postc:  Se cap < volume della tanica restituisce vero, falso altrimenti
*/
    { assert(t);
    assert(cap >0);
    if ( cap < (t->alt*t->larg*t->prof) ) {
                t->cap_max = cap;
                return TRUE;
                }
          else
                return FALSE;
          }

int AggLiq( TanicaP t, double tot )
/* aggiunge liquido a una tanica, se la disponibilità lo consente
prec: : t != NULL && tot >0
postc: restituisce vero se  tot è stato aggiunto alla tanica, falso altrimenti.
*/
{assert(t);
 assert(tot >0);
 if ( (t->quant_corr + tot) <= (t->cap_max) )
          {t->quant_corr = t->quant_corr + tot;
          return TRUE;
          }
          else
          return FALSE;
}

int PrelLiq( TanicaP t, double tot )
/*  preleva tot di liquido da una tanica, se tot è minore del liquido contenuto
prec: t != NULL && tot>0
postc: restituisce vero se  tot di liquido è stato prelevato, falso altrimenti*/
    {assert(t);
      assert(tot >0);
      if ( t->quant_corr >= tot )
           {t->quant_corr = t->quant_corr - tot;
          return TRUE;
          }
          else
          return FALSE;
  }
```