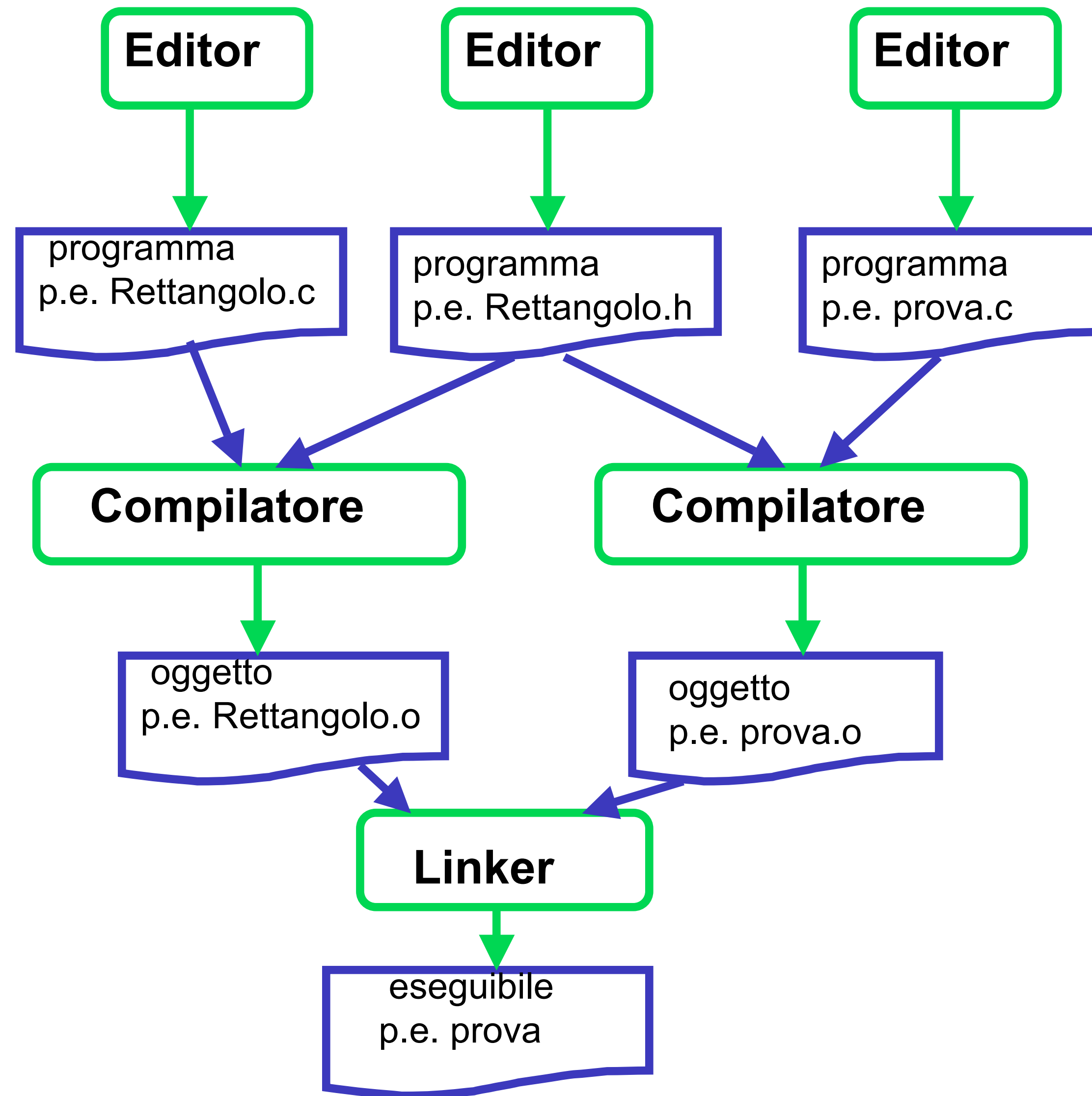
Dai files del programma al codice eseguibile



Sotto Unix o Linux

```
cc -c mio.c
```

-c è un opzione che dice al compilatore di fermarsi dopo aver prodotto il codice oggetto corrispondete al codice C nel file di testo mio.c

Se tutto è andato bene il compilatore ha prodotto un file mio.o.

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

l'opzione -o annuncia il nome del file eseguibile

esempio Rettangolo:

```
cc -c rettangolo.c
cc -c prova.c
cc -o prova prova.o rettangolo.o
```

infine per eseguire il programma si digita semplicemente

```
prova
```

OBIETTIVO CORRETTEZZA: specificazione completa con precondizioni e postcondizioni

- **Le precondizioni e postcondizioni aggiunte al prototipo, consentono di fornire una specificazione precisa della funzione**

- **Le precondizioni stabiliscono le eventuali restrizioni di definizione della funzione sull'insieme dei dati in input (dato dal tipo dei parametri)**

  il chiamante deve assicurarsi di chiamare la funzione solo se le precondizioni sono soddisfatte, infatti se una precondizione non è rispettata al momento della chiamata, la funzione può comportarsi in modo imprevedibile.

- **Le postcondizioni specificano l'output della funzione.**

Se il chiamante ha rispettato le precondizioni, la funzione deve rispettare le postcondizioni, cioè restituire i valori in conformità con le postcondizioni.

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

```
void alt( RettangoloP r, double alt );
/*
Prec: r!= NULL
Postc: restituisce l’altezza*/

void alt( RettangoloP r, double alt );
/*
Prec: nessuna
Postc: se r !=NULL restituisce l’altezza, altrimenti 0*/
```

Nome file: Rettangolo.h - Specificazione della classe Rettangolo

```
void ModAlt( RettangoloP r, double alt );
/* Modifica l’altezza
Prec: r!= NULL && alt >0 */

double Larg( RettangoloP r );
/* prec: r !=NULL
Postc: Restituisce la larghezza */

void ModLarg( RettangoloP r, double larg);
/* Modifica la larghezza */
Prec: r!= NULL && larg >0 */

double Perimetro( RettangoloP r );
/* prec: r!= NULL
Postc: restituisce il perimetro */

double Area( RettangoloP r );
/* prec r!= NULL
Postc: restituisce l’area */
```

Nome file: Rettangolo.h - Specificazione della classe Rettangolo

```
/**** Nome della classe ****/
typedef struct rettangolo *RettangoloP;

/**** Metodi per operare sulla classe ****/

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


/* Distruttore*/
void DistrRettangolo( RettangoloP r );

double Alt( RettangoloP r );
/*prec: r != NULL
* Postc: Restituisce l'altezza */
```

Esempio: sia L una collezione e x un elemento nell'universo di definizione di L, consideriamo la funzione

$$\texttt{cerca}(L,x) = \begin{cases} 1 \text{ se } \exists x\ x \in L \text{ (in genere abbreviato se } x \in L) \\ 0 \text{ altrimenti} \end{cases}$$

Qual'è il risultato di `cerca` se L = ∅?

Dal punto di vista logico `cerca`(∅,x) = 0, infatti ∃x x ∈ ∅ è evidentemente falsa

OBIETTIVO CORRETTEZZA: specificazione attraverso precondizioni e postcondizioni

Supponendo di aver rappresentato in memoria la collezione come una lista, l’implementazione è:

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

{if  (!L) return 0;
 if(L->elem == el) return 1;
 return cerca(L->nextPtr, el); }
```

“cerca” è una funzione totale

CONVENZIONE:
Ometteremo il riferimento alle precondizioni se non ce ne sono

OBIETTIVO CORRETTEZZA: specificazione attraverso precondizioni e postcondizioni

Ma in qualche caso è necessario distinguere il caso di ricerca con insuccesso dal caso dell'insieme vuoto.

La prima soluzione proposta esclude l'insieme vuoto come argomento della funzione:

```
int cerca1(Listptr L, Elem el)
/* verifica se l'elemento el occorre almeno una volta nella collezione
*prec: L != NULL
*postc:  restituisce 1 se l'elemento occorre nella lista,  0 altrimenti */

{if (L->elem == el) return 1;
 if (L -> nextPtr) return cerca1(L->nextPtr, el);
/* Non ci devono essere chiamate su L==NULL */
  else return 0; }
```

"cerca1" è una funzione parziale

OBIETTIVO CORRETTEZZA: specificazione attraverso precondizioni e postcondizioni

La seconda soluzione proposta distingue in output i due casi:

```
int cerca2(Listptr L, Elem el)
/* verifica se l'elemento el occorre (almeno una volta) nella collezione
*postc: se L = ∅ restituisce -1, altrimenti se  l'elemento e' presente
 restituisce 1 e in caso contrario 0.*/

{if (!L)  return -1;
if (L->elem == el) return 1;
if (L -> nextPtr) return cerca2(L->nextPtr, el);
else return 0;
}
```

“cerca2” è una funzione totale

OBIETTIVO CORRETTEZZA: specificazione attraverso precondizioni e postcondizioni

E' importante capire che le precondizioni non richiedono alcuna verifica nel codice della funzione, anzi spesso sono introdotte proprio 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!**

leggibilità attraverso precondizioni e postcondizioni

```
struct nodo {
 int elem;
 struct nodo * next;
};
typedef struct nodo  Lista;
typedef  Lista* ListaPtr;
```

```
 void insert (ListaPtr *L, int val)
{ListaPtr new;
if (*L == NULL || (*L)->elem >= val)
   {new = malloc(sizeof(Lista));
    new->elem = val;
    new->next = *L;
    *L = new;
    }
  else insert (&(*L)->next, val);
}
```

```
void insert (ListaPtr *L, int val)
{/*prec: La lista deve essere ordinata in ordine crescente*/
 /*postc: inserisce val nella prima posizione consistente con l'ordine */
ListaPtr new;
if (*L == NULL || (*L)->elem >= val)
    {new = malloc(sizeof(Lista));
     new->elem = val;
     new->next = *L;
     *L = new;
     }
   else insert (&(*L)->next, val);
}
```

OBIETTIVO LEGGIBILITA': precondizioni e postcondizioni, ma anche NOMI AUTOESPLICATIVI.

```
 void insListOrd (ListaPtr *L, int val)
{/*prec: La lista deve essere ordinata in ordine crescente*/
 /*postc: inserisce val nella prima posizione consistente con l'ordine */
ListaPtr new;
if (*L == NULL || (*L)->elem >= val)
    {new = malloc(sizeof(Nodo));
     new->elem = val;
     new->next = *L;
     *L = new;
     }
   else insListOrd(&(*L)->next, val);
}
```

Un programma si scrive una volta, ma si legge e si usa molte volte!