Programmazione Robusta

I programmi robusti si comportano bene anche in circostanze eccezionalmente avverse, un po' come i ponti che non crollano alla prima inondazione. Questo si ottiene introducendo nel codice delle istruzioni che consentono di verificare alcune condizioni di correttezza. Così facendo si riduce di gran lunga il tempo di testing perchè se l'errore si verifica è il programma stesso a segnalarlo. Se invece un errore è rilevato all'uscita dal programma o per esempio si ottiene un messaggio "Segmentation fault" (che si ottiene sotto Unix ogni volta che i programma tenta di accedere a una locazione di memoria che non è stata allocata per il programma), bisogna prima cercare l'errore (nel caso dell'esempio in quale istruzione si è verificato questo tentativo di accesso illegale) che naturalmente si può correggere solo una volta trovato. Se invece si introducono delle istruzioni che prevedono gli errori e producono un opportuno messaggio che consente di localizzarlo, si risparmia molto tempo.

Un tipico errore

E' un errore molto comune tentare di accedere a un puntatore nullo, questo errore porterà a un "Segmentation fault". Questo può essere **prevenuto** come nell'esempio seguente:

```
int  mia_funzione(  ... )
{
/*si introduce una variabile r di tipo puntatore a
una struttura */
...
if ( r == NULL )
        {printf("mia_funzione: errore puntatore
nullo!\n");
     return ... ;   /* restituiamo un valore
opportunamente scelto*/
qui le istruzioni che usano r

}
...
}
```

Ma questo approccio applicato a ogni puntatore implicato rende il codice abbastanza pesante! Si può introdurre un controllo da disabilitare quando si è sicuri che l'intero programma che usa

questo metodo non chiama la funzione su un puntatore nullo. Questo può essere realizzato utilizzando la compilazione condizionata.

```
int  mia_funzione(  ... )
{
/*si introduce una variabile r di tipo puntatore a
una struttura */
...
#ifdef FASE_TESTING
if ( r == NULL )
  {printf("mia_funzione: errore puntatore
nullo!\n");
     return ... ;   /* restituiamo un valore
opportunamente scelto*/
      }
#endif
qui le istruzioni che usano r
...
}
```

Nella fase di testing si inserisce l'istruzione:

```
#define FASE_TESTING
```

in un header file che viene importato e la si rimuove quando si è sicuri che il programma è corretto. In questo modo le istruzioni di controllo vengono eseguite solo nella fase di testing dell'intero programma.

Un metodo fornito dal C: l'assert

Si può ottenere un risultato analogo a quello di prima ma più semplice usando la macro assert della libreria standard del C. La macro assert prende in input un'espressione booleana e non fa nulla se questa è vera, altrimenti ferma l'esecuzione del programma invocando exit(1), che segnala l'errore. Bisogna importarne la definizione nel file "assert.h" :

```
#include <assert.h>


int  mia_funzione(  ... )
{/*si introduce una variabile r di tipo puntatore a
una struttura */
```

```
...
assert(r)

qui le istruzioni che usano r
...
}
```

L'implementazione di assert varia nei dettagli dell'informazione che può fornire ma un tipico messaggio è:

```
Assertion(r) failed in "esempio.c", function
"mia_funzione", line 57
```

L'uso di assert rende il codice robusto senza appesantirlo.

Più di una condizione da verificare

Un primo uso di assert riguarda le **precondizioni**. Per esempio, sia mia_funzione una funzione che ha come parametri un puntatore a una funzione c, e due interi p1 e p2, inoltre la funzione sia definita solo per valori positivi di p1 e p2, che deve anche non eccedere un valore massimo. Si ha il seguente prototipo

```
int mia_funzione ( StructP c, int p1, int p2 );
```

e le condizioni sui parametri diventano precondizioni:

```
/*prec: (c != NULL) && (p1 > 0) && (p2 >= 0) && (p2
<= MAX_P) */
```

Per poter controllare, in fase di testing, il corretto uso di questa funzione si può inserire una chiamata di assert sulla precondizione.

```
#include <assert.h>

int mia_funzione( StructP c, int p1, int p2 )
{
/*eventuali dichiarazioni di variabili*/

assert( (c != NULL) && (p1 > 0) &&  (p2 >= 0) && (p2
```

```
<= MAX_P));

  ... /* si fa qualche cosa su c */
  }
```

Questo, in caso una delle condizioni fallisse, darebbe origine al seguente messaggio

```
Assertion  ((c != NULL) && (p1 > 0) && ( p2 >= 0) &&
(p2 <= MAX_P)) failed in "esempio.c",
function "mia_funzione", line 348
```

Così il programma è robusto, fornisce una autodiagnosi, ma non dice quale delle condizioni è fallita, quindi conviene effettuare una chiamata di assert su ciascuna condizione, in modo da individuare l'errore.

```
int mia_funzione( StructP c, int p1, int p2 )
{
  assert( c != NULL );
  assert( p1 > 0 );
  assert( p2 >= 0);
  assert(p2 <= MAX_P);

... /* si fa qualche cosa su c */
  }
```

Oppure si può pensare di fare una chiamata di assert per ogni parametro

```
#include <assert.h>

int mia_funzione( StructP c, int p1, int p2 )
{
  assert( c != NULL );
  assert( p1 > 0 );
  assert( (p2 >= 0) && (p2 <= MAX_P) );

  ... /* si fa qualche cosa su c */
  }
```

Una volta individuato un errore nel parametro p2, in caso di necessità, si può ricompilre il programma spezzando la condizione.

Questa riduzione delle chiamate di assert rende il programma più veloce ma non drasticamente, come si deduce esaminandone l'implementazione:

```
#ifndef NDEBUG
    #define assert(x)
        {if (!(x))
            printf("Assertion %s failed in file %s,
        at line %d \n,#x,__FILE__,__LINE__);
        exit(1);
        }
#else
    #define assert(x)
#endif
```

Cioè se NDEBUG **non** è definito allora sostituisci, ovunque occorra, assert(x) con

```
{if (!(x))
printf("Assertion %s failed in file %s, at line
%d\n, #x,__FILE__,__LINE__);
exit(1);
}
```

se invece NDEBUG è definito non si sostituisce niente a assert(x), cioè la si rimuove dal programma. Non facendo nulla quindi l'assert viene implementata e il controllo su x viene effettuato. Quando il programma viene considerato sufficientemente controllato, basta definire NDEBUG, ricompilare e rilinkare il programma. Sotto Unix si può compilare con un'opzione, come segue

```
cc -DNDEBUG prog.c
```

il cui effetto è quello di predefinire il simbolo NDEBUG. Questo è equivalente a introdurre in un header file che viene incluso in tutti i file di codice.

```
#define NDEBUG
```

Usando assert quindi si ottengono più risultati in uno:

- robustezza (individuazione degli errori mediante autodiagnosi)

- codice snello (le verifiche con assert vengono eliminate quando non servono più)

- facilità di lettura (l'uso di uno strumento standard)