

Obiettivo robustezza

Un tipico errore:

riferimento a un puntatore nullo

Prima soluzione.

```
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 il codice è appesantito!

Obiettivo robustezza

Un tipico errore:

riferimento a un puntatore nullo

Seconda soluzione: uso della compilazione condizionata.

```
#define FASE_TESTING

...

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*/
            }

#ifdef FASE_TESTING
qui le istruzioni che usano r
...
    }
```

COMPILAZIONE CONDIZIONATA

I comandi che possono essere usati nella la compilazione condizionata sono:

#if include un qualche testo in dipendenza del valore di un'espressione costante

#else include un qualche testo, se i precedenti test in **#if** , **#ifdef** , **#ifndef** o **#elif** sono falliti

#elif facilita la lettura di if annidati, l'ultimo deve essere **#else**

#endif conclude l'espressione condizionale

#ifdef include un qualche testo se un nome di macro é definito

#ifndef include un qualche testo se un nome di macro non é definito

COMPILAZIONE CONDIZIONALE

SINTASSI

#if *espressione costante 1*
gruppo di linee di codice 1
#elif *espressione costante 2*
gruppo di linee di codice 2
...
#elif *espressione costante n*
gruppo di linee di codice n
#else
ultimo gruppo di linee di codice

SEMANTICA

Se *espressione costante* $\neq 0$ verrà compilato il *gruppo di linee di codice 1* e tutto il resto viene scaricato altrimenti verrà valutata l' *espressione costante 2* del successivo **#elif** e se diversa da 0 verrà compilato il *gruppo di linee di codice 2* e il resto scaricato e così via, se nessuna delle *espressione costante i* dà un valore diverso da 0, allora verrà compilato *ultimo gruppo di linee di codice* e verrà ignorato il resto

COMPILAZIONE CONDIZIONATA

SINTASSI

#if *espressione costante*
gruppo di linee di codice 1
#else
gruppo di linee di codice 2

SEMANTICA

Se espressione costante $\neq 0$ verrà compilato il gruppo di linee di codice 1 e l'altro viene scaricato altrimenti verrà compilato il gruppo di linee di codice 2 e l'altro scaricato

Un' *espressione costante* deve essere valutata a tempo di compilazione e quindi può essere solo di tipo intero, coinvolgendo solo costanti intere, costanti di tipo char e lo speciale operatore define. Tutta l'aritmetica è fatta usando il tipo long o unsigned long.

Un *gruppo di linee di codice* può contenere un qualsiasi numero di linee di testo, anche altre direttive di preprocessore o anche nessuna linea

Obiettivo robustezza

Un tipico errore:

riferimento a un puntatore nullo

Terza soluzione: uso di assert.

```
#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  
    ...  
  }
```

Se `r==NULL` il programma viene interrotto e a video si leggerà il seguente messaggio (o uno molto simile):

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

Una possibile definizione di assert(x):

```
#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 alcunchè ad assert(x), cioè la si rimuove dal programma.

Uso di assert e Precondizioni

```
int mia_funzione ( StructP c, int p1, int p2 );  
/*prec: (c != NULL) && (p1 > 0) && (p2 >= 0) && (p2 <=  
MAX_P) */
```

```
#include <assert.h>
```

```
int mia_funzione( StructP c, int p1, int p2 ) {  
    assert( (c != NULL) && (p1 > 0) && (p2 >= 0) && (p2  
<= MAX_P));  
  
    ... /* si fa qualche cosa su c */  
}
```

Ottenendo il seguente messaggio diagnostico in caso di violazione di una delle condizioni:

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



```
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 */
}
```

Ottenendo il seguente messaggio diagnostico in caso di violazione della condizioni su p1:

```
Assertion (( p1 > 0))
failed in "esempio.c",
function "mia_funzione", line 348)
```

```
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 */
}
```

Ottenendo il seguente messaggio diagnostico in caso di violazione di una delle condizioni su p2:

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

Se necessario, si ricompila separando le due condizioni

Usando assert quindi si ottengono più risultati in uno:

- 1. robustezza (individuazione degli errori mediante autodiagnosi)**
- 2. codice snello (le verifiche con assert vengono eliminate quando non servono più)**
- 3. facilità di lettura (l'uso di uno strumento standard)**