www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Nota importante sulle condizioni composte nelle istruzioni C

Alcuni tipi di istruzioni C (istruzioni condizionali, cicli, ...) dipendono dalla valutazione di opportune *condizioni*. Tali condizioni possono essere di tipo *composto*, ovverosia ottenuto componendo (con gli operatori *!*, ||, &&) due o più condizioni componenti. Le condizioni composte, se utilizzate senza la necessaria attenzione, possono essere fonte di problemi.

Qui di seguito vengono descritti due casi in cui è importante scegliere oculatamente tra diversi modi di esprimere una stessa condizione, nonostante *in termini formali* tali modi appaiano del tutto equivalenti. Vedremo infatti, attraverso esempi, come la modalità di esecuzione dei programmi C da parte del calcolatore sia tale che condizioni formalmente equivalenti possono portare, nella pratica, a programmi che si comportano in modo molto differente.

Esempio 1: lettura di un dato che può trovarsi oltre l'ultimo elemento di un array

Supponiamo di dover scrivere un ciclo *while* che esamina gli elementi di un array di MAX elementi interi. Il ciclo termina quando viene trovato un elemento positivo, oppure quando è stato esaminato l'ultimo elemento dell'array (quello avente indice MAX-1). Dette Cont una variabile intera usata come contatore e Arr la variabile array, il codice che implementa il ciclo sarà quindi

```
Cont = 0;
while ( (Cont<MAX) && (Arr[Cont]<=0) )
{
   ++Cont;
}
```

E' importante notare, a questo punto, che per quanto le condizioni

```
( (Cont<MAX) && (Arr[Cont]<=0) )
```

e

```
( (Arr[Cont]<=0) && (Cont<MAX) )
```

siano equivalenti dal punto di vista formale, esse *non sono affatto intercambiabili* nel contesto del codice C sopra considerato. Infatti scrivere tale codice nella forma

```
Cont = 0;
while ( (Arr[Cont]<=0 ) && (Cont < MAX) )
{
   ++Cont;
}
```

sarebbe un grave errore.

Il motivo è il seguente. E' possibile che, durante l'esecuzione del programma, Cont venga incrementato fino a raggiungere il valore MAX. In queste condizioni il ciclo termina (ovvero non viene eseguita una nuova iterazione) in entrambi i frammenti di codice alternativi sopra riportati. Tale terminazione viene decisa dal calcolatore in base al valore di verità della condizione di esecuzione del ciclo. Tale condizione, pertanto, viene valutata dal calcolatore anche quando Cont ha raggiunto il valore MAX. Questa operazione di valutazione, per quanto necessaria, presenta tuttavia il rischio di produrre un errore di *segmentation fault*: infatti essa richiede di prelevare dalla memoria il dato Array[Cont] che, per Cont==MAX, corrisponde a Array[MAX]. Il problema è che l'elemento di indice MAX dell'array Arr *non esiste*, giacché i valori ammissibili arrivano solo fino a MAX-1! Tale elemento corrisponde in effetti alla zona di memoria immediatamente successiva all'ultimo elemento dell'array, che non fa parte dell'array. Se tale zona non appartiene al programma, il sistema operativo rileva una violazione degli spazi di memoria (segmentation fault) e termina il programma.

Fortunatamente il C mette a disposizione del programmatore un meccanismo che rende possibile l'esecuzione del codice senza tuttavia rischiare la terminazione forzata del programma. Infatti in C la valutazione delle condizioni composte (come quella considerata) viene svolta tenendo presente il seguente vincolo di esecuzione:

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Si consideri il processo di di valutazione di una condizione composta. Se, durante il suo svolgimento, i valori delle condizioni componenti già valutate sono tali da rendere noto il valore della condizione composta prima ancora di valutare le condizioni componenti restanti, queste ultime non vengono valutate affatto.

Questo vincolo risponde a criteri di efficienza esecutiva (perché valutare condizioni il cui valore è ininfluente per il valore della condizione composta?) e contemporaneamente fornisce al programmatore un meccanismo per evitare problemi del tipo sopra descritto.

Ora si capisce, infatti, perché i due frammenti di codice sopra riportati non siano equivalenti. Nel caso del primo, quando si verifica il caso limite Cont==MAX la seconda condizione componente (ovverosia `(Array[Cont]<=0)`) non viene valutata affatto: dunque la lettura dell'area di memoria corrispondente ad Array[MAX] non avviene. Nel caso del secondo frammento di codice, invece, quando Cont==MAX la condizione `(Array[Cont]>0)` viene valutata: dunque la zona di memoria corrispondente ad Array[MAX] viene letta ed esiste il rischio di *segmentation fault*.

Esempio 2: lettura o meno di un dato puntato da un puntatore avente valore NULL.

Supponiamo di dover elaborare dati aventi la forma di *struct*, al cui interno sia presente un campo chiamato CampoIntero. Per accedere a tali dati usiamo un puntatore chiamato Punt, che di volta in volta viene fatto puntare alllo specifico dato (ovvero alla specifica *struct*) da elaborare. Se non esiste un dato pronto per l'elaborazione, a Punt viene dato il valore NULL.

Immaginiamo che l'elaborazione del dato debba avvenire solo se il contenuto del suo campo CampoIntero è negativo. Allora, una volta definito il valore di Punt, inseriremo nel programma un frammento del tipo seguente:

```
if ((NULL != Punt) && (Punt -> CampoIntero < 0))
{
  ... elaborazione del dato ...
}
```

La condizione `((NULL != Punt) && (Punt -> CampoIntero < 0))` è composta da due condizioni componenti: `(NULL != Punt)` e `(Punt -> CampoIntero < 0)`, ed è formalmente equivalente a quella `((Punt -> CampoIntero < 0) && (NULL != Punt))`. Tale equivalenza formale significa che le due condizioni hanno identico valore in qualsiasi situazione *in cui risulta possibile valutarne il valore*.

Nel funzionamento di un programma reale, però, esistono situazioni in cui una condizione non può essere valutata. Nel caso di questo esempio, quando Punt vale NULL, la valutazione della condizione `(Punt -> CampoIntero < 0)` non è possibile: se eseguita, genera un errore in fase di esecuzione del programma che interrompe l'esecuzione. Infatti è impossibile estrarre il campo CampoDati dal dato puntato da Punt, visto che Punt -avendo valore NULL- per definizione non punta ad alcun dato. In un caso simile l'esecuzione del programma viene terminato con un errore di *segmentation fault*.

Per questa ragione riscrivere il precedente frammento di programma ponendo la condizione nella forma (formalmente equivalente ma per nulla equivalente in termini di comportamento reale)

```
if ((Punt -> CampoIntero < 0) && (NULL != Punt))
  {
    ... elaborazione del dato ...
  }
```

sarebbe un grave errore.

Questo tipo di problema non può verificarsi se il frammento di programma è scritto nel primo modo: infatti le condizioni utilizzate nel primo e nel secondo frammento, per quanto formalmente equivalenti, portano a modalità differenti di esecuzione del programma. Ciò dipende dal vincolo di esecuzione sulla valutazione delle condizioni composte introdotto nell'Esempio 1. Tale vincolo fa sì che nel caso in cui Punt vale NULL il primo frammento di programma venga eseguito senza errori. La prima condizione componente ad essere valutata è infatti, in quel caso, `(NULL != Punt)`: dal momento che essa è falsa, la seconda condizione componente `(Punt -> CampoIntero < 0)` non viene valutata poiché `(false && X)` ha sempre valore falso, qualunque sia il valore di verità di `X`.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari