Restrict: Endgame – come, quando e perché usare il type qualifier restrict in C

Scott Lang/Ant-Man: *Voi avete mai studiato la fisica quantistica?*
Natasha Romanoff/Vedova Nera: *Solo per fare conversazione.*

Beh, vediamo un po': siamo ad Agosto, fa caldo, sono in ferie… *macchimmelofaffare* di scrivere un post? E vabbè, lo faccio, ma sarà un post leggero e frivolo, in cui riutilizzerò la struttura, il tema e perfino il titolo **dell'ultimo articolo** che ho scritto. E quindi mi ispirerò ancora al formato tutto trame e sottotrame del bel **Avengers: Endgame** per fare una dotta (?) disquisizione nonché *"chiusura finale"* su un altro misterioso e *(fortunatamente)* poco usato **type qualifier** del **C**: **restrict**.



…ti spiego: restrict non serve a niente…
Per seguire lo stesso schema dell'ultimo articolo bisogna ora dare le notizie, e direi di cominciare con quella buona:

- La buona notizia è che, similmente a **volatile** che si usava solo col contagocce, **restrict** si usa ancora meno, visto che è veramente poco utile (e nella citazione di chiusura dell'articolo capirete il perché) ma, comunque, non è vietato usarlo.

E anche qui c'è una cattiva notizia collegata a quella buona:

- Visto che usare **restrict** è possibile, se lo si usa bisogna usarlo bene, altrimenti si possono avere comportamenti indefiniti (*undefined behavior*) durante l'esecuzione del programma: *macchittelofaffare?*

E veniamo al dunque: nel titolo di questo articolo preannunciavo la descrizione del come, quando e perché usare **restrict** in C, per cui ora ci tocca cominciare con:

IL COME:

Ricordiamo che **restrict** è un **type qualifier**, e quindi si usa (solo) per aggiungere una proprietà a un **pointer** (ripeto, solo a un **pointer**) a un tipo qualsiasi già esistente, quindi il come è semplicissimo e, ad esempio, possiamo scrivere:

```
int *dummy; // un int pointer normalissimo che si chiama dummy
int *restrict res_dummy; // un int pointer di tipo restrict che si chiama res_dummy
```

E vi ricordo nuovamente che i **type qualifier** sono solo quattro: **volatile**, **restrict**, **const** e **_Atomic**, quindi ce ne mancano ancora due da analizzare… (*uff, siamo solo a metà!*). E ora siamo pronti a passare al prossimo punto:

IL QUANDO:

Per capire il quando bisogna descrivere brevemente cosa fa **restrict** applicato a un *pointer*: dice al compilatore che l'oggetto puntato da quel *pointer* sarà puntato esclusivamente attraverso quel *pointer* e mai con un altro: quindi **restrict** si dovrebbe usare quando abbiamo il controllo totale del codice

scritto e siamo sicuri che questa condizione di uso esclusivo sarà sempre rispettata (*e se no: undefined behaviour!*). Uhm, *"controllo totale del codice scritto"* questo è possibile per programmatori molto esperti (**ne ho già parlato**) però diventa molto difficile lavorando in team (anche piccoli).

E ora siamo pronti per il prossimo punto, che è:

IL PERCHÉ:

Il perché direi che si può riassumere in: *perché voglio ottenere un codice super-ultra-stra-mega efficiente e ottimizzato e, visto che non mi fido delle ottimizzazioni realizzate autonomamente dal compilatore, voglio aiutarlo un po'…* beh, questo appare un po' presuntuoso e pretenzioso e, tra l'altro, **ne ho già parlato qui**.

Riassumendo, a questo punto possiamo trarre la inevitabile conclusione, che è: **restrict** non è del tutto inutile, ma è da usare con tale parsimonia e attenzione che si può benissimo farne a meno, e non a caso in alcuni ambienti di programmazione molto restrittivi (ad esempio quelli in cui si usano le norme **MISRA**) l'uso è **scoraggiato**.

È però una buona abitudine usare **restrict** quando si scrivono funzioni molto ottimizzate (magari di libreria) con uso di argomenti *"limitati"* (ma con la condizione di **documentare adeguatamente** agli utilizzatori le restrizioni d'uso). Ad esempio, se ho bisogno di scrivere una mia versione in **C** della **memcpy()**, perché non voglio/posso usare quella standard, dovrei dichiararla come:

```
void *myMemcpy(void *restrict dest, const void *restrict src, size_t n);
```

Così la nuova funzione avrebbe esattamente le stesse limitazioni e lo stesso prototipo della **memcpy()** standard, dove gli argomenti *src* e *dest* non possono "*overlappare*".

E adesso, dopo tante parole, ci vuole un po' di codice, per

cui vi propongo un esempio semplicissimo, che è uno dei tanti possibili (l'ho scopiazzato da **cppreference.com** perché non avevo voglia di scriverne uno ad-hoc):

```
// copy() - funzione di copia senza overlap void copy(int n, int *restrict dest, int *restrict src)
{ while (n-- > 0) *dest++ = *src++; // nessuno degli oggetti modificati tramite *dest // è uguale a nessuno degli oggetti letti tramite // *src. Il compilatore è libero di ottimizzare, // vettorializzare, mappare le pagine, ecc.
} // funzione main
int main()
{ extern int d[100]; copy(50, d + 50, d); // OK copy(50, d + 1, d); // Undefined behavior: d[1] è utilizzato attraverso // entrambi dest e src in copy()
}
```

E per concludere, vi aggiungo la citazione (che ho già usato **qui** e **qui**) **di uno molto più bravo di me**, come avevo promesso all'inizio dell'articolo. Una citazione che riassume quello che penso di **restrict**, e che può aiutare a tornare coi piedi per terra quando ci si perde in dettagli ininfluenti e si scrive del pessimo codice pensando di scrivere un capolavoro:

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming" [Donald Ervin Knuth, "Computer Programming as an Art", 1974]

E per oggi penso che possa bastare. Rilassatevi, che siamo in Agosto!

Ciao, e al prossimo post!

https://italiancoders.it/restrict-endgame-come-quando-e-perche-usare-il-type-qualifier-restrict-in-c/