

# The Last Keyword: `_Generic`, `noreturn` e `static_assert` – come usare le nuove keyword del C – pt.3

*Pierre d'Alençon: La tua decenza sarà la tua fine.*

*Jacques Le Gris: E non c'è fine alla mia decenza.*

Ebbene sì: dopo una sofferta riflessione ho deciso di scrivere l'ultimo atto sulle (*relativamente*) nuove *Keyword* del nostro amato C. L'avevo promesso! E più volte! E quindi era diventata una questione d'onore, per cui come non citare un bel film che fa dell'onore il suo tema principale? **The Last Duel** è un'altra gemma che si aggiunge alle altre che ci ha regalato il grandissimo **Ridley Scott**. Certo non è un film allegro, quindi non sono riuscito a trovare una citazione molto brillante... ma questo è un momento così, questi ultimi anni non sono stati esattamente un periodo per frasi brillanti. *Ma noi programmatori siamo fiduciosi per natura! Speriamo bene...*



**...lotterò fino all'ultima keyword, lo giuro!...**

Allora, riepiloghiamo: nella **prima parte** abbiamo parlato di **complex** e **imaginary** (citando di striscio anche **bool**, **thread\_local** e **atomic** di cui avevamo già parlato in passato); nella **seconda parte** abbiamo parlato di **alignas** e **alignof**. Quindi, per completare l'opera, ci mancano solo **\_Generic**, **\_Noreturn** e **\_Static\_assert**, che sono state introdotte col **C11** e che saranno l'argomento di questa terza (ed ultima) parte. Anche per queste *keyword* vale la solita regola descritta in questa auto-citazione:

*...un po' come è successo, per esempio, con le variabili booleane **\_Bool** definite nell'header **stdbool.h**: questa norma del nome che inizia con **underscore+maiuscola** è seguita per tutte le nuove keyword del C, che possono, poi, essere usate con l'header corrispondente per "raddrizzare" il nome...*

e, quindi, le nostre nuove *keyword* le useremo come **noreturn** e **static\_assert** usando gli appositi *header* standard, meno **\_Generic** che è l'eccezione che conferma la regola: non ha una macro di semplificazione (ce ne faremo una ragione...).

Ok, allora cominciamo proprio con **\_Generic**: grazie a questa *keyword* si può usare una sorta di "*programmazione generica*" nel C, una possibilità che era anteriormente assente, e che ricorda un po' i meccanismi di *overloading* (e un po' anche quelli dei *template*) tipici del C++. La sintassi è la seguente:

**\_Generic**(espressione\_di\_controllo, lista\_diAssociazione)  
espressione\_di\_controllo: una espressione di assegnamento che rappresenta un tipo  
lista\_diAssociazione: una lista di associazione con la seguente sintassi: tipo: espressione, tipo: espressione, ..., default: espressione

Ok, ammettiamolo: questo dice poco, anzi dice che anche un linguaggio stabile, concreto e senza fronzoli come il C diventa astruso quando tenta di aggiungere qualche costrutto

**C++like** (sarà il karma...). Ma forse se trasformiamo quanto sopra in un semplice esempio tutto sarà più chiaro:

```
#include <stdio.h> // prototipi locali
int intAdd(int a, int b);
float floatAdd(float a, float b);
void errAdd(); // la mia funzione di addizione generica
#define genAdd(x, y) \ _Generic((x), int: intAdd(x, y), float:
floatAdd(x, y), default: errAdd()) // main - funzione main
int main(int argc, char const *argv[])
{ // set variabili di test int ia = 1; int ib = 2; float fa =
1.1; float fb = 2.2; double da = 1.3; double db = 2.4; //
eseguo le somme generiche genAdd(ia, ib); genAdd(fa, fb);
genAdd(da, db); return 0;
} // intAdd - somma due int
int intAdd(int a, int b)
{ printf("%d + %d = %d\n", a, b, a + b); return a + b;
} // floatAdd - somma due float
float floatAdd(float a, float b)
{ printf("%.1f + %.1f = %.1f\n", a, b, a + b); return a + b;
} // errAdd - nessuna somma per i tipi passati
void errAdd()
{ printf("errore di tipo\n");
}
```

Sì, così va meglio. Seguendo l'esempio si possono eseguire mille varianti per gli usi più svariati (e ricordarsi di rispettare la sintassi: *default* è opzionale, non si possono ripetere tipi, e un po' di altri dettagli che si trovano nel manuale). Io francamente non uso mai **\_Generic**, ma usarlo non è una cattiva idea. Ah, il codice qui sopra produce il seguente risultato:

```
1 + 2 = 3
1.1 + 2.2 = 3.3
errore di tipo
```

E passiamo a **noreturn**. È una sorta di *type specifier* che vale solo per le funzioni, quindi è, a tutti gli effetti, un *function specifier*. La sintassi è questa:

`_Noreturn` dichiarazione\_di\_funzione oppure, usando l'include apposito: `#include <stdnoreturn.h>`  
`noreturn` dichiarazione\_di\_funzione

e a che serve? Serve per dire che la funzione non avrà un ritorno. Questo non vuol dire che non ha un valore di ritorno, o meglio, che non ritorna nulla (*per quello c'è già void!*), ma vuol dire proprio che non ritorna! Ok, facciamo un esempio (*anzi, due: crepi l'avarizia!*): la system call `_exit(2)` e la funzione libc `exit(3)` sono due esempi ben calzanti di cosa vuol dire **noreturn** e non necessitano di ulteriori spiegazioni. E quindi quando si esegue una funzione di questo tipo il flusso del codice si ferma, e le successive istruzioni non vengono eseguite. Ok, forse vi sto confondendo le idee, meglio proseguire con un esempio:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdnoreturn.h> // prototipi locali
noreturn void okAbort();
noreturn void nokAbort(); // main() - funzione main
int main(void)
{ printf("sto per eseguire abort()...\n"); okAbort();
  printf("questa scritta non sarà mai mostrata!\n");
} // okAbort() - esegue "cose" e quindi un abort()
noreturn void okAbort()
{ // faccio cose // ... // chiamo la abort(3) abort(); // ok:
  se scrivo codice dopo NON viene eseguito // potrei scrivere
  codice (ma tanto non verrà mai eseguito!) // ...
} // nokAbort() - esegue "cose" e quindi un abort()
condizionato
noreturn void nokAbort(int i)
{ // faccio cose // ... // chiamo la abort(3) if (i > 0)
  abort(); // nok: eseguo solo se i > 0 // se i <= 0 faccio cose
  ma non va bene: è una funzione noreturn! // E quindi avrò dei
  comportamenti non prevedibili (undefined behavior) // e anche
  un bel Warning in compilazione! // ...
}
```

Compilando ed eseguendo otterremo questo risultato:

sto per eseguire abort()...  
Annullato (core dump creato)

Spero che l'esempio (*con i soliti ampi commenti*) abbia chiarito le idee... è necessario aggiungere, però, che **noreturn** è un po' come **restrict** (*rileggere l'articolo, please*) e anche un po' come **volatile** (*altro articolo da rileggere*): è uno *specifier* da usare con cautela, perché se usato male può dare comportamenti non prevedibili (*undefined behavior*). Con **noreturn** ci impegniamo con il compilatore a scrivere una funzione che farà quanto promesso: "*non ritornare*", e quindi il compilatore farà le opportune ottimizzazioni tenendo in conto questo. Ma se non manteniamo la promessa (è il caso di *nokAbort()* dell'esempio mostrato prima) avremo comportamenti imprevedibili, quindi occhio! Notare che nelle librerie di sistema le funzioni **abort(3)**, **exit(3)** ecc. sono state ridefinite come **noreturn** (*perché lo sono veramente!*).

Dai, ci siamo quasi. Vedo la luce in fondo al tunnel: parliamo di **static\_assert** e la finiamo qui.

Cosa è **static\_assert**? È, come dice il nome, una **assert** statica (e grazie al...). Ok, spieghiamoci: sicuramente conoscete già la macro **assert(3)** che consente di eseguire dei controlli a *run-time* su delle istruzioni, abortendo l'esecuzione del programma se la asserzione fallisce. È abbastanza frequente l'uso in fase di *debug* per verificare cose tipo divisioni per 0, radici di numeri negativi, ecc. Ed è già predisposta per essere disabilitata per le versioni *release* dei programmi (*uhm... io non uso quasi mai la assert(3), ma è una scelta molto personale, ammetto che è una macro molto utile*). E la **static\_assert**? È molto simile, ma funziona a *compile-time* invece che a *run-time* (e quindi è statica). La sintassi è:

```
_Static_assert(espressione, messaggio) oppure, usando  
l'include apposito: #include <assert.h>  
static_assert(espressione, messaggio)
```

Inserendo delle **static\_assert** "*mirate*" nel codice possiamo,

già durante la compilazione, renderci conto se c'è qualcosa che non va. E a questo punto ci vuole un bell'esempio. Vai col codice!

```
#include <assert.h> // main - funzione main
int main(int argc, char const *argv[])
{ // le mie asserzioni static_assert(sizeof(int) == 4, "codice
per sistemi con int da 4 bytes"); static_assert(sizeof(long)
== 4, "codice per sistemi con long da 4 bytes"); // faccio
cose // ... return 0;
}
```

Ecco, quello nell'esempio è un classico (ma non l'unico) uso della **static\_assert**: quando si compila codice su varie piattaforme si possono aggiungere delle asserzioni in maniera che se la piattaforma non è compatibile già a livello di compilazione provochiamo uno stop in compilazione con relativo messaggio di errore: questo ci eviterà futuri grattacapi a *run-time* (*"ma come, su quell'altra macchina funzionava benissimo!"*). Ad esempio il codice appena mostrato, sulla mia macchina (Linux su amd64) produce questo durante la compilazione:

```
In file included from staticassert.c:1:
staticassert.c: In function 'main':
staticassert.c:8:5: error: static assertion failed: "codice
per sistemi con long da 4 bytes" 8 |
static_assert(sizeof(long) == 4, "codice per sistemi con long
da 4 bytes"); | ^~~~~~
```

la prima asserzione fila liscia come l'olio, mentre la seconda blocca la compilazione. Semplice, no?

E vabbè, dai, per oggi può bastare. *Finalmente ho compiuto la mia vecchia promessa di ultimare l'argomento "nuove keyword" e ora mi sento veramente più leggero...*

Ciao, e al prossimo post!

[https://italiancoders.it/the-last-duel-\\_generic-noreturn-e-static\\_assert-come-usare-le-nuove-keyword-del-c-pt-3/](https://italiancoders.it/the-last-duel-_generic-noreturn-e-static_assert-come-usare-le-nuove-keyword-del-c-pt-3/)