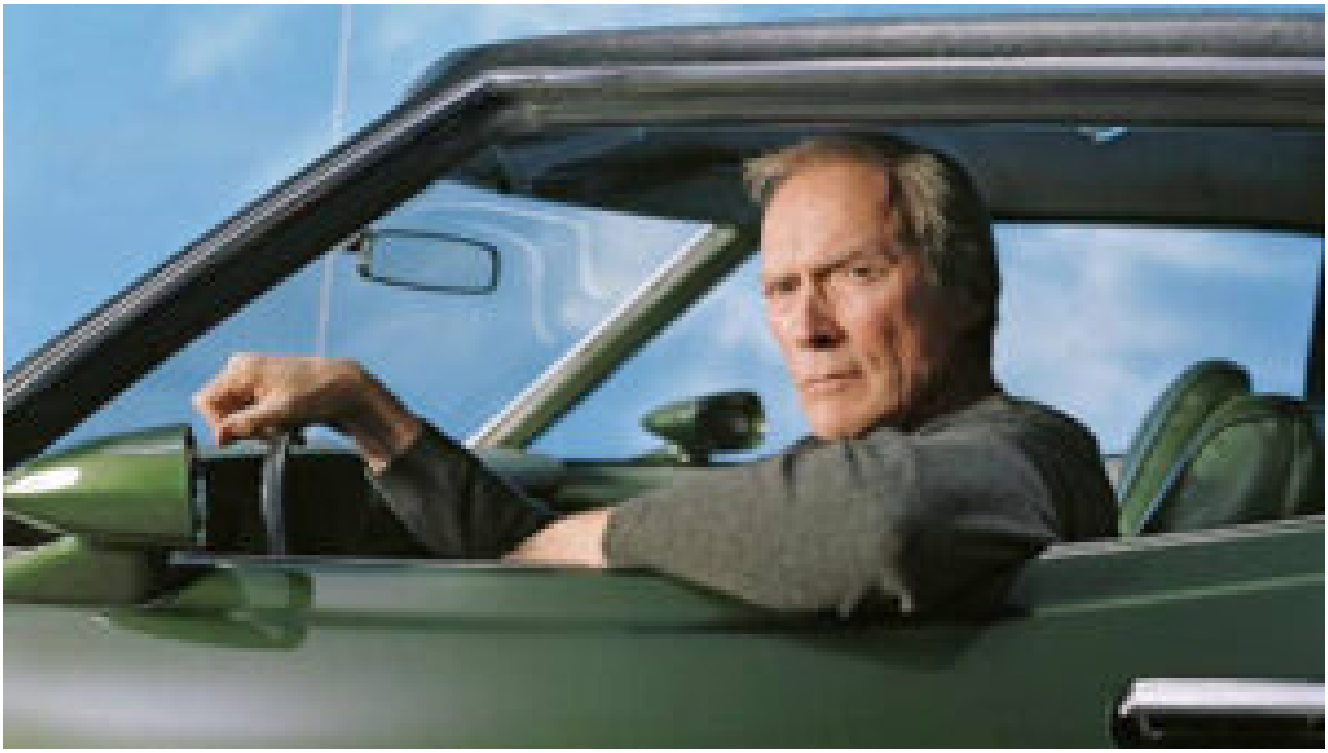
Gran Alignment: alignas e alignof – come usare le nuove keyword del C – pt.2

Padre Janovich: *Cosa posso fare per lei, Mister Kowalski?*
Walt Kowalski: *Sono venuto a confessarmi.*
Padre Janovich: *O Dio del Cielo, che cosa ha fatto?*
Walt Kowalski: *Niente, veda di non scaldarsi.*
Padre Janovich: *Che cosa sta per fare?*
Walt Kowalski: *Me la fa questa confessione o no?*

Dopo aver celebrato l'ultima fatica del grande **Clint Eastwood** (**Cry Macho**), rimarremo sullo stesso tema e andremo indietro di qualche anno per celebrare un altro gran film: **Gran Torino**. Qui Clint è **Walt Kowalski**: un vecchio brontolone con un cuore d'oro. Un tipo preciso e rigoroso con una gran passione per la sua splendida "**Ford Gran Torino**" (*e guai a toccargliela*).

Ecco, per spostarci sul nostro amato **C** possiamo dire che anche usare due delle nuove keyword del **C11**, **_Alignas** e **_Alignof**, è indice di grande precisione e rigore in programmazione, anche se in realtà (e, forse, anche per fortuna) l'uso è abbastanza limitato ad alcuni casi particolari (ad esempio la programmazione embedded super-ottimizzata). Comunque, anche se di uso limitato sono keyword utili e, come minimo, bisogna sapere cosa sono e a che servono, perché non si sa mai…



…ho appena rifatto l’allineamento delle ruote. Stai alla larga!…

Allora, vediamo un po’. **_Alignas** e **_Alignof** sono due (relativamente) nuove keyword che tratteremo in coppia: questa è un po’ una forzatura considerando che fanno cose abbastanza diverse, ma, visto che operano sullo stesso argomento (l’allineamento), permettetemi la forzatura (*grazie*). Ok andiamo, cominciamo con **_Alignas**:

Il type specifier _Alignas

_Alignas è un *type specifier* (come **_Complex** e **_Imaginary**, visti nella **parte 1 dell’articolo**) che si può applicare a tipi ed espressioni. Anche in questo caso è valida la norma **già citata** del nome che comincia con *undescore+maiuscola* per facilitare la retro-compatibilità (come già spiegato nell’articolo precedente e che vi invito a rileggere, please). E quindi, anche **_Alignas** deve essere, opportunamente, usato attraverso il suo *header stdalign.h*, che gli permette di trasformarsi, magicamente, nel nome **alignas** (*che è molto più user-friendly, no?*). E a cosa serve **alignas**? Ebbene, come si intuisce dal nome, serve ad *allineare*. Ma allineare cosa? Beh, per non perderci in cervellotiche spiegazioni farò direttamente un bell’esempio col codice, che, spero, renderà

bene l’idea:

```
#include <stdio.h> struct struct_a { char pippo; float pluto;
short paperino;
}; struct struct_b { float pluto; short paperino; char pippo;
}; int main(void)
{ printf("sizeof char : %zu\n", sizeof(char)); printf("sizeof
short: %zu\n", sizeof(short)); printf("sizeof float: %zu\n",
sizeof(float)); printf("il size di struct_a è: %zu\n",
sizeof(struct struct_a)); printf("il size di struct_b è:
%zu\n", sizeof(struct struct_b));
}
```

se compilate ed eseguite questo codice il risultato (*con GCC su Linux su amd64*) sarà:

```
sizeof char : 1
sizeof short: 2
sizeof float: 4
il size di struct_a è: 12
il size di struct_b è: 8
```

Che sorpresa! Vi dirò, questa non dovrebbe essere una sorpresa ma, mio malgrado, ho scoperto che non pochi programmatori (specialmente quelli non avvezzi alla programmazione *low-level*) si sorprendono. Allora, senza tentare di spiegare come funzionano la ruota e l’acqua calda, riassumerò (molto brevemente e in maniera semplificata) come funziona l’allineamento nel **C** (*e chi lo sa già può, ovviamente, saltare a piè pari le prossime righe*):

1. La dimensione di una struttura è condizionata dall’allineamento dei dati usato dal compilatore (a sua volta condizionato dalla macchina: per questo nel test qui sopra ho specificato compilatore, OS e CPU): l’allineamento è sempre una potenza del 2, e l’elemento più grande di una struttura (nel nostro caso il *float* con size 4) condiziona l’allineamento di tutta la struttura.
2. Visto il punto 1, si può affermare che la dimensione

minima della nostra struttura è il multiplo di 4 più vicino alla somma dei membri (nel nostro caso la somma è 7, quindi nel caso ottimale abbiamo ottenuto 8).
3. Il compilatore aggiunge, se necessario, dei byte di *padding* per raggiungere il *size* finale (nel caso di *struct_b* ha aggiunto un *byte*). Se mettiamo i membri *a caso* il *padding* può essere notevole (nel caso di *struct_a* abbiamo 5 *byte* di *padding*).
4. Un buon ordine a prova di *padding* è quello in cui gli elementi adiacenti possono essere raggruppati in sottogruppi di (in questo caso) 4 *byte*: in *struct_b pluto* occupa un sottogruppo e *paperino+pippo* stanno in un altro sottogruppo, quindi il risultato è ottimizzato. Lo stesso non si può dir di *struct_a* dove *pippo* non può stare nello stesso sottogruppo di *pluto* (e quindi occuperà da solo 4 *byte*) e idem per *pippo*.

Evidentemente questo comportamento *imprevisto* delle strutture può comportare vari problemi, alcuni di efficienza (dati non ordinati e non ridotti al minimo non vengono ben digeriti dalla CPU) e altri problemi, più gravi, di comportamenti indesiderati: ad esempio, trasferendo dati binari tra macchine diverse è possibile che, scrivendo i dati in una struttura destinazione, non si possa accedere correttamente ai membri perché le due macchine hanno allineamenti diversi. Anticamente questo si risolveva aggiungendo dei membri *dummy* per allineare forzatamente la struttura (ed evitando il *padding* del compilatore), ma ora ci viene in aiuto **alignas**! (*sospiro di sollievo*).

E come si usa **alignas**? Ecco, vi propongo un esempio *scemo* (*ma proprio scemo*) che fa una cosa un po' controproducente ma rende bene l'idea:

```
#include <stdio.h>
#include <stdalign.h> struct struct_a { char pippo; float pluto; short paperino;
}; struct struct_b { float pluto; short paperino; char pippo;
```

```
}; struct struct_c { alignas(4) float pluto; alignas(4) short paperino; alignas(4) char pippo;
}; int main(void)
{ printf("il size di struct_a è: %zu\n", sizeof(struct struct_a)); printf("il size di struct_b è: %zu\n", sizeof(struct struct_b)); printf("il size di struct_c è: %zu\n", sizeof(struct struct_c));
}
```

E il risultato dell'esecuzione sarà:

```
il size di struct_a è: 12
il size di struct_b è: 8
il size di struct_c è: 12
```

Ecco, grazie a questo codice abbiamo ottenuto di trasformare la nostra splendida e allineata *struct_b* in una struttura disallineata (*o meglio: allineata come vogliamo noi*): questo serve a poco, ma indica come con **alignas** possiamo modellare i dati (con un *padding* forzato) per avere un comportamento sicuro anche, ad esempio, nel trasferimento tra macchine diverse. Ovviamente si può fare un uso molto più intelligente e sofisticato di **alignas**, ma spero che l'esempio abbia chiarito le idee.

Ah, dimenticavo: **alignas** ha delle limitazioni d'uso (relative all'uso con variabili *register* e *bit-fields*, ecc.) ma per la lista completa vi rimando alle pagine del **manuale**. E adesso è il turno di **_Alignof**:

L'operatore _Alignof

_Alignof è un *operatore*, esattamente come **sizeof** con cui condivide il modo d'uso e una certa assonanza del nome. Anche **_Alignof** può essere, opportunamente, usato attraverso il suo *header stdalign.h*, che gli permette di trasformarsi (di nuovo magicamente) in **alignof**. Come si può intuire dal nome, **alignof** ritorna l'allineamento del tipo (o oggetto) passato in argomento, esattamente come **sizeof** ritorna il *size*. Quindi, grazie a questo nuovo operatore, possiamo controllare bene gli

allineamenti degli oggetti che usiamo, e agire opportunamente in base alle necessità: risulta evidente che il campo applicativo è lo stesso di **alignas** (con cui, non a caso, condivide l'*header*), e un semplice esempio d'uso potrebbe essere il seguente:

```
struct struct_a { char pippo; float pluto; short paperino;
}; struct struct_b { float pluto; short paperino; char pippo;
}; struct struct_c { alignas(4) float pluto; alignas(4) short
paperino; alignas(4) char pippo;
}; struct struct_d { alignas(8) float pluto; alignas(8) short
paperino; alignas(8) char pippo;
}; int main(void)
{ printf("il size di struct_a è: %zu\n", sizeof(struct
struct_a)); printf("il alignof di struct_a è: %zu\n",
alignof(struct struct_a)); printf("il size di struct_b è:
%zu\n", sizeof(struct struct_b)); printf("il alignof di
struct_b è: %zu\n", alignof(struct struct_b)); printf("il size
di struct_c è: %zu\n", sizeof(struct struct_c)); printf("il
alignof di struct_c è: %zu\n", alignof(struct struct_c));
printf("il size di struct_d è: %zu\n", sizeof(struct
struct_d)); printf("il alignof di struct_d è: %zu\n",
alignof(struct struct_d));
}
```

e il risultato dell'esecuzione sarà:

```
il size di struct_a è: 12
il alignof di struct_a è: 4
il size di struct_b è: 8
il alignof di struct_b è: 4
il size di struct_c è: 12
il alignof di struct_c è: 4
il size di struct_d è: 24
il alignof di struct_d è: 8
```

ossia: grazie ad **alignof** possiamo verificare le condizioni base di allineamento e possiamo anche verificare il risultato di una operazione di **alignas**. Semplice no? Anche in questo caso l'esempio era un po' scemo, ma spero che renda bene l'idea.

E con questo credo che sia tutto. Nel prossimo articolo tratteremo le restanti nuove *keyword* del **C11**, e per oggi può bastare: *con tutti questi allineamenti e disallineamenti mi è venuto un mal di testa… e spero di non averlo fatto venire anche a voi!*

Ciao, e al prossimo post!

https://italiancoders.it/gran-alignment-alignas-e-alignof-come-usare-le-nuove-keyword-del-c-pt-2/