Librerie header-only? Forse! – come scrivere una libreria header-only in C

Holland March: *Alla fine nessuno si è fatto male.*
Jackson Healy: *A me sembra di sì...*
Holland March: *Nel senso che sono morti in fretta, non è che si sono fatti male.*

Ebbene si, oggi parleremo di un argomento abbastanza inusuale e poco conosciuto del **C** (almeno per quello che mi riguarda): le librerie **header-only**. Come vedremo più avanti questo è un argomento un po' controverso e (forse) di dubbia utilità, ma per evitare gli strali dei fan (ce ne sono) di questo tipo di librerie ho deciso che questo sarà un articolo del tipo **"Forse!"** invece che un **"No, grazie!"** (*a proposito, di* ***"No, grazie!"*** *ne ho scritti un po', li potete trovare* ***qui***, ***qui***, ***qui***, ***qui***, ***qui*** *e* ***qui***, *mentre un esempio di* ***"Forse!"***, *o meglio, era uno* ***"Scusate"***, *lo trovate* ***qui***). L'argomento dell'articolo, tra il serio e il faceto, si intona con il bel film **The Nice Guys** del bravo **Shane Black**, un film brillante di quelli che Hollywood anticamente sfornava a ripetizione, ma che ora non sanno quasi più fare…



…secondo me è una libreria header-only…

E cosa sono le queste librerie **header-only**? Ok, partiamo dalle basi (**Linux**, eh! Ma anche in altri S.O. funziona più o meno allo stesso modo): qui stiamo per parlare di librerie di sviluppo (su **Linux,** famiglia **Deb,** sono i pacchetti **-dev*), che sono quelle che servono per realizzare una applicazione. Normalmente (anzi canonicamente) una libreria di sviluppo si distribuisce (semplificando un po') attraverso due file:

- Un *header-file* (un **.h*) che contiene i prototipi delle funzioni disponibili più la definizione di eventuali strutture dati tipiche della libreria: questo permette di scrivere e compilare una applicazione che usa quello che mette a disposizione la libreria.
- Un *lib-file* (un **.a* o un **.so*), che è una versione compilata dei file di implementazione (i **.c*) che compongono la libreria. Questo *lib-file* può avere formato a link statico (**.a*) o dinamico (**.so*): grazie a questo si può *linkare* ed eseguire la applicazione che usa la libreria. Quindi quando si distribuisce la applicazione in formato eseguibile bisogna anche distribuire la libreria associata.
- La libreria da distribuire avrà varie versioni in base a tipo/versione del S.O. della macchina su cui si vuole installare e usare.

Come si può intuire (*e se no che lo scrivo a fare questo articolo?*) c'è anche una maniera *"non canonica"* di distribuzione, ed è proprio quella delle librerie **header-only**:

- Si distribuisce un solo file, un *header-file* un po' strano che contiene prototipi, strutture dati e anche implementazioni! Questo *header* si deve includere in tutti i file del progetto che usano funzioni e/o strutture dati della libreria.
- Si compila e via! Si può eseguire l'applicazione *linkando* solo le (eventuali) altre librerie necessarie all'applicazione. Quindi quando si distribuisce la applicazione in formato eseguibile non c'è bisogno di

distribuire anche la libreria associata.

- La portabilità è buona: una libreria **header-only** è, a tutti gli effetti, un file sorgente, quindi se si riesce a compilare sarà perfettamente compatibile con la macchina che stiamo usando (eventuali problemi verranno dopo, per distribuire la sola applicazione precompilata ed eseguibile).
- Notare che alcune delle librerie **header-only** disponibili in rete usano alcuni trucchi (basati sulla definizione di alcune macro ad-hoc) per far si che uno solo dei file **.c* (del progetto che userà la libreria) includa la parte di implementazione del *header-file*, mentre gli altri **.c* includeranno solo la lista dei prototipi contenuta nell'*header-file*. In questo caso si dovrebbe parlare di librerie **pseudo-header-only**, che non sono l'argomento di questo articolo: qui parleremo solo di quelle *"vere"*.

Apparentemente questo metodo di distribuzione è interessante, ma non è oro tutto quello che luccica: in letteratura tecnica sono riportati alcuni difetti:

- In un grande progetto con molti sorgenti **.c* che includono la libreria **header-only** una modifica a quest'ultima provocherà un notevole allargamento dei tempi di compilazione (l'*header* verrà ricompilato *n-volte*). Con la potenza delle macchine attuali questo potrebbe non essere un gran problema (anche se non bisogna sottovalutarlo: alcuni progetti sono veramente grandi e comprendono migliaia di sorgenti).
- Per evitare gli ovvi errori di ridefinizione in compilazione si fa grande uso (come vedremo più avanti) di *storage classes* di tipo **static** e/o **static inline**: questo provoca una crescita notevole del codice macchina generato, un problema difficilmente ottimizzabile dal compilatore.

Due problemi da niente, no?

E vabbé, è ora di dare un esempio reale (ultra-semplificato) di un progetto che usa una libreria **header-only**; il progetto include:

- **mylib.h**: la libreria **header-only** (è una *"vera"*, senza i trucchi citati sopra).
- **test1.c**: un sorgente **C** che definisce una funzione **fun1()** che usa internamente una funzione della libreria **mylib.h**.
- **test2.c**: un sorgebte **C** che definisce una funzione **fun2()** che usa internamente una funzione della libreria **mylib.h**.
- **test.c**: un sorgente con un **main()** che chiama le funzioni definite in **test1.c** e t**est2.c**.

Vai col codice!

```
// mylib.h - una libreria header-only
#include <stdio.h> #ifdef STATICINLINE
#define MY_API static inline
#define MY_API_STR "static inline api"
#elif defined STATIC
#define MY_API static
#define MY_API_STR "static api"
#elif defined INLINE
#define MY_API inline
#define MY_API_STR "inline api"
#elif defined EXTERN
#define MY_API extern
#define MY_API_STR "extern api"
#else
#define MY_API
#define MY_API_STR "api"
#endif // libfun1 - una funzione generica della libreria
MY_API void libfun1(void)
{ printf(
} // libfun2 - una funzione generica della libreria
MY_API void libfun2(void)
{ printf(
}
```

```
// test1.c - modulo per il test della libreria header-only mylib
#include "mylib.h" void fun1(void)
{ // chiamo una funzione della libreria header-only mylib
libfun1();
}

// test2.c - modulo per il test della libreria header-only mylib
#include "mylib.h" void fun2(void)
{ // chiamo una funzione della libreria header-only mylib
libfun2();
}

// test.c - main di test della libreria header-only mylib // test - funzione main
int main(void)
{ extern void fun1(void); extern void fun2(void); // chiamo le funzioni dei moduli test1 e test2 fun1(); fun2(); return 0;
}
```

Come sempre il codice è **ben commentato**, però in questo caso qualche spiegazione è doverosa: a parte le espressioni condizionali iniziali (che vedremo più avanti) il meccanismo di avere prototipi e definizioni in **mylib.h** è semplice e abbastanza chiaro, no ? Quindi visto che **test1.c** e **test2.c** includono **mylib.h** possono chiamare, rispettivamente, **libfun1()** e **libfun2()** internamente alle funzioni globali **fun1()** e **fun2()**. Dopodiché **test.c**, che non ha bisogno di includere **mylib.h**, chiama le funzioni globali **fun1()** e **fun2()**: tutto abbastanza lineare.

Ma l'inclusione in più sorgenti di uno strano *header-file* con prototipi e implementazioni non è triviale, e quindi bisogna scegliere accuratamente le *storage classes* delle funzioni, perché gli errori di ridefinizione sono dietro l'angolo. Il meccanismo che permette il funzionamento è, come anticipato sopra, quello che si vede nelle prime linee di **mylib.h**, dove ho messo alcune espressioni condizionali che permettono di testare e mostrare le varie maniere operative. Poi, una volta

capito il meccanismo, **mylib.h** si può semplificare togliendo tutti i condizionali e lasciando solo il tipo di *storage classes* scelto.

Vediamo, allora, cosa succede compilando (e, se possibile, eseguendo) la nostra applicazione:

```
aldo@Linux $ gcc test.c test1.c test2.c -o test -DSTATICINLINE
aldo@Linux $ ./test
static inline api: sono libfun1
static inline api: sono libfun2 aldo@Linux $ gcc test.c test1.c test2.c -o test -DSTATIC
aldo@Linux $ ./test
static api: sono libfun1
static api: sono libfun2 aldo@Linux $ gcc test.c test1.c test2.c -o test -DINLINE
/usr/bin/ld: /tmp/ccdLwOwW.o: in function `fun1':
test1.c:(.text+0x9): undefined reference to `libfun1'
/usr/bin/ld: /tmp/ccK47RcM.o: in function `fun2':
test2.c:(.text+0x9): undefined reference to `libfun2'
collect2: error: ld returned 1 exit status aldo@Linux $ gcc test.c test1.c test2.c -o test -DEXTERN
/usr/bin/ld: /tmp/ccNPBHJP.o: in function `libfun1':
test2.c:(.text+0x0): multiple definition of `libfun1';
/tmp/cczGXRP8.o:test1.c:(.text+0x0): first defined here
/usr/bin/ld: /tmp/ccNPBHJP.o: in function `libfun2':
test2.c:(.text+0x33): multiple definition of `libfun2';
/tmp/cczGXRP8.o:test1.c:(.text+0x33): first defined here
collect2: error: ld returned 1 exit status aldo@Linux $ gcc test.c test1.c test2.c -o test
/usr/bin/ld: /tmp/cc5t0uT9.o: in function `libfun1':
test2.c:(.text+0x0): multiple definition of `libfun1';
/tmp/ccLox6Tj.o:test1.c:(.text+0x0): first defined here
/usr/bin/ld: /tmp/cc5t0uT9.o: in function `libfun2':
test2.c:(.text+0x33): multiple definition of `libfun2';
/tmp/ccLox6Tj.o:test1.c:(.text+0x33): first defined here
collect2: error: ld returned 1 exit status
```

Come si nota il programma si compila ed esegue regolarmente solo nei modi **STATICINLINE** e **STATIC**: la parola chiave è, in entrambi i casi **static**, perché definendo così le funzioni la

definizione resta limitata al singolo file che include **mylib.h**, quindi non ci sarà nessun errore di ridefinizione (ma si cade nel problema descritto sopra: una crescita del codice macchina risultante). Aggiungendo a **static** anche il *function specifier* **inline**, il risultato non cambia: funziona bene e, inoltre, il compilatore cerca, se possibile, di "*inlineare*" la funzione (e anche in questo caso si cade nello stesso problema descritto sopra, però l'eseguibile potrebbe essere più efficiente). Notare che le funzioni mostrano anche il tipo di interfaccia **MY_API** in uso, ottenuta dalla macro **MY_API_STR** di **mylib.h**.

E cosa succede compilando coi modi **INLINE**, **EXTERN** e "*senza modo*"? Succede che ci sono errori di compilazione, quindi non possiamo neanche eseguire. Nel modo **INLINE** (senza **static**) notiamo che la funzione non viene resa disponibile al *linker,* e quindi abbiamo degli errori del tipo "*undefined reference to libfun1*": il *linker* ha bisogno, per questo tipo di librerie della parola magica **static**. Nei modi **EXTERN** e "*senza modo*" abbiamo, invece, degli errori del tipo "*multiple definition of libfun1*", che indicano che, chiedendo un *linkaggio* di tipo **extern** (o senza tipo) e fornendo poi la funzione ogni volta che si include **mylib.h** si verifica il problema di avere multiple definizioni. Come previsto.

È tutto chiaro? Spero di si.

E qui ci sta bene aggiungere qualche considerazione personale, perché non ho ancora detto se considero buone o cattive le librerie **header-only** (*magari l'ho fatto intuire, però*): devo ammettere che non mi piacciono, perché le considero una forzatura del linguaggio: il fatto che, con le dovute accortezze, si riesca a farle digerire al compilatore e al *linker,* non significa che sia una buona idea usarle.

E poi inserire la definizione di una funzione dentro un *header-file* è più roba da **C++** (**il lato oscuro della forza**): lì è abbastanza usuale che in un *header-file* ci sia, all'interno

della definizione di una classe, anche il codice dei metodi (*questo è Ok ma non mi piace: io preferisco scriverlo nel file di implementazione della classe*); per non parlare poi dei (famigerati) *template*, dove è addirittura obbligatorio (o perlomeno molto raccomandabile) avere *"tutto"* (*class template* e codice dei metodi) in un solo *header-file*. Anche per questo i compilatori **C** e **C++** trattano in maniera un po' differente le *storage classes* descritte sopra. Conclusione: se proprio vi piacciono le librerie **header-only** passate al **C++**, *ah ah ah.*

Per oggi può bastare, sono contento di avere scritto un nuovo articolo del tipo **"Forse!"**, perché su alcuni argomenti non è necessario essere troppo radicali… ma su altri si, quindi aspettatevi qualche altro **"No, grazie!"** in futuro! *E non trattenete il respiro nell'attesa, mi raccomando!*

Ciao, e al prossimo post!

https://italiancoders.it/librerie-header-only-forse-come-scrivere-una-libreria-header-only-in-c/