C++ Perfect Forwarding



In questo articolo parleremo di una importantissima funzionalità del C++ introdotta con lo standard 11 del linguaggio: Il cosidetto *perfect forwarding* degli argomenti.
Quando si chiede ad uno sviluppatore C++ quali siano le caratteristiche salienti dello standard 11, novantanove su cento risponde con: Move Semantic, Smart Pointers, *auto* e *lambdas*.
Per qualche motivo, quasi mai, si annovera il p*erfect forwarding* degli argomenti tra le grandi innovazioni del C++ standard 11.
Cerchiamo quindi di dare un po' di visibilità a questa (forse) poco conosciuta, feature dell'Olimpo del moderno C++.

UN oscuro SENTIRE

Morpheus, in Matrix, nella celebre scena in cui siede in poltrona, faccia a faccia con Neo, elenca una serie di metafore per indurre il futuro Eletto ad intravedere una realtà differente rispetto a quella in cui ha sempre vissuto.

Morpheus:

“[..] L’ avverti quando vai al lavoro, quando vai in chiesa, quando paghi le tasse [..] ”

Il programmatore C++; ma direi, in misura molto maggiore il programmatore C++ dell’era *pre-std11*, ha da sempre avvertito che qualcosa non andava in certi usi della *Standard Template Library*.
Qualcosa di oscuro e di non ben definito, specialmente quando usavi certe funzioni o certi metodi.
Ti lasciava lì a pensarci su per qualche minuto, chiedendoti se c’era qualcosa di sbagliato in quello che avevi appena scritto; se per caso, non esistesse un modo migliore per fare quello che avevi fatto.

push_back

Prendiamo per esempio un più che *innoquo* stralcio di codice come quello dell’esempio che segue:

```
//a simple vector holding strings
std::vector<std::string> vs; //push back a string at the end of the vector vs.push_back("Pizzeria Bella Napoli");
```

Sembra semplice, direi banale; ma non c’è qualcosa che ci lascia con un retrogusto amaro?
Se state pensando che ci sia qualcosa che non va nel tipo dell’arogmento: c-string literal, applicato ad un container instanziato per il tipo: std::string, beh, non è quello il punto.
Proviamo ad andare a sbirciare dentro l’header: <vector> e diamo un occhiata a come è dichiarato il metodo: push_back di: std::vector<T>.
Notiamo che il metodo è *overloaded*:

```
//from the C++11 Standard template <class T, class Allocator = allocator<T>>
class vector {
public:
.. void push_back(const T& x); //push back lvalue
void push_back(T&& x); //push back rvalue ..
```

```
};
```

push_back, a partire dallo standard 11 è stato sovraccaricato (*overloaded*) per *references* di tipo *lvalue* (std98) e per *references* di tipo *rvalue* (std11).
Quando il compilatore incontra la chiamata: vs.push_back("Pizzeria Bella Napoli") vede una discrepanza tra il tipo dell'*argomento* passato: const char[22] e il tipo del *parametro* della push_back (una *reference* a std::string).
Poichè std::string possiede un costruttore con un parametro di tipo: c-string e poichè questo costruttore non è marcato explicit, il compilatore può effettuare per noi una conversione automatica.
Tratterà pertanto il codice come se lo avessimo scritto così:

```
//a temporary std::string object is created
vs.push_back(std::string("Pizzeria Bella Napoli"));
```

A conti fatti, questo codice compila, questo codice gira; e a fine giornata, tutti se ne vanno a casa felici e contenti.
Tutti eccetto il programmatore C++ che questo codice lo ha scritto.
Siccome, come tutti i programmatori C++ che si rispettino, è coscienzioso, si è fatto qualche domanda scomoda.
Dopo averci pensato un po' su infatti, si è reso conto che il codice non è così efficiente come invece dovrebbe essere.
Nella sua testa si è visualizzato tutti i passaggi che avverranno a runtime:

1. Si crea una stringa temporanea inizializzata con la c-string "Pizzeria Bella Napoli"
2. La stringa così creata è un rvalue, quindi il compilatore ha potuto scegliere la versione della push_back che prende T&&.
3. Costruisce quindi una std::string nella memoria interna del vector vs usando il costruttore di *move*, passando come argomento la stringa temporanea creata al punto 1.

Quindi, sta pensando il programmatore, il suo codice non sta

chiamando una volta sola il costruttore di std::string, no, lo sta chiamando due volte.
Non solo, prima che la push_back ritorni al chiamante, sarà stato chiamato anche il distruttore di std::string della stringa temporanea.
Il programmatore non è per nulla soddisfatto di questo comportamento; gli sembra che per una cosa molto semplice, a runtime, si stiano facendo decisamente *troppe* cose.
Se solo esistesse un modo per prendere la c-string e si potesse applicare direttamente alla chiamata del costruttore nel passo 3) si eviterebbe del tutto la costruzione e la distruzione dell'oggetto temporaneo.

Ma non ci salva il costruttore di move?

Un'osservazione che a questo punto potrebbe essere sollevata è che il costo della costruzione della std::string nel passo 3) avviene chiamando il costruttore di *move* e non quello di *copia*.
Le costruzioni per *move*, si sa, sono impiegate proprio per minimizzare i costi di costruzione degli oggetti in presenza di argomenti *rvalues*.
Questo è sicuramente vero, però, in generale, non possiamo sapere a priori quanto davvero costi chiamare il costruttore di un oggetto, anche fosse quello di *move.*
Quello che sarebbe invece auspicabile, sarebbe di poter chiamare una volta sola il costruttore dell'oggetto *target*, *direttamente sopra la memoria interna allocata dal container*.
I programmatori C++ che si sono formati nell'era dello standard 98 si sono da sempre scontrati con questa fastidiosa mancanza.
Non solo, nello standard 98, non esiste la *move*, esiste solo la *copia*, quindi il problema diventa molto più serio.
Un bel giorno però, è arrivato lo Standard 11 e fortunatamente, in dote, ha portato la soluzione a questo pernicioso problema.

EMPLACE_back

Dallo standard 11, ogni standard container che supporta la push_back, è stato arricchito anche del metodo emplace_back.
Wow, ma come funziona? Semplice, il nostro esempio con la push_back si trasforma in:

```
//a simple vector holding strings std::vector<std::string> vs;
//construct a string at the end of the vector, in-place.
vs.emplace_back("Pizzeria Bella Napoli");
```

Dal punto di vista della quantità di codice da scrivere non è cambiato molto, cambia solo il nome del metodo chiamato.
Dal punto di vista della sostanza però, cambia tutto, completamente!
Con emplace_back non si crea alcun oggetto temporaneo, nemmeno uno.
Ciò che avviene invece, è che la c-string viene passata *direttamente* al costruttore di std::string applicato alla *memoria privata* allocata nello std::vector.
Questo, era esattamente quello che avremmo desiderato fare sin dall'inizio.
Ma c'è di più: se avessimo voluto utilizzare il costruttore di *fill* di std::string che prende un char e ripete per *n* volte quel carattere, avremmo scritto:

```
//a simple vector holding strings std::vector<std::string> vs;
//construct a string using std::string fill constructor at the end of the vector, in-place. vs.emplace_back(10, 'x');
```

Il vector contiene adesso un elemento con valore "xxxxxxxxxx".
Vorrei farvi notare che con push_back, non è più possibile che il compilatore applichi per noi dei trucchi di coversione automatica:

```
//a simple vector holding strings std::vector<std::string> vs;
//I want to use std::string fill constructor.. vs.push_back(10, 'x'); //ugh.. compilation fails, no way to convert T& with 2 arguments of type int and char.. //so if we want to use push_back we must write explicitly:
```

```
vs.push_back(std::string(10, 'x')); //ok
```

A questo punto dovremmo aver intuito che emplace_back è decisamente molto diverso da push_back.
Il metodo push_back possiede un unico parametro di tipo reference: const T& oppure T&&.
Il metodo emplace_back invece, accetta un *numero variabile* di argomenti e li passa esattamente come li ha ricevuti, "*it perfect forward them*", ad un costruttore di T.
Dicevamo che ogni standard container che supporta push_back adesso supporta anche emplace_back.
Oltre a questo, ogni standard container che supporta il metodo insert ora supporta anche il metodo emplace.
I container associativi possiedono anche un metodo emplace_hint che accetta un "*hint iterator*" così come avviene per il metodo insert.

perchè "perfect" forwarding ?

Chi a questo punto si sentisse soddisfatto e appagato della spiegazione può tranquillamente sospendere la lettura, andare ad usare le funzioni *emplace* e vivere felice.
Chi invece fosse più curioso e volesse vedere cosa c'è sotto al cofano prosegua la lettura.
Ciò che segue, serve ovviamente per chi volesse scrivere le proprie funzioni *perfect forwarding*.
Bene, allora proseguiamo.
Perchè lo hanno chiamato *perfect forwarding* degli argomenti e non solamente *forwarding* degli argomenti?
Non stiamo semplicemente facendo l'equivalente di prendere degli argomenti (in numero variabile) e di passarli ad una funzione "*target*"?
No, non è così semplice, la spiegazione di quello che c'è dietro al *perfect forwarding* degli argomenti in C++ è più articolata e complessa.
Proviamo allora a dire a parole quello che fa il *perfect forwarding*.
Il *perfect forwarding* rende possibile scrivere funzioni

template – "*forwarding*" – che prendono un numero arbitrario di argomenti e li "*forwardano*" ad altre funzioni di modo che queste funzioni "*target*" ricevano esattamente gli *stessi* argomenti E con le stesse "*proprietà*" espresse al momento della chiamata della funzione "*forwarding*".
Eh..?? Va bene, lo ammetto, forse nemmeno il Conte Raffaello Mascetti di Amici Miei avrebbe osato cotanta *supercazzola*; ma non vi preoccupate, coraggio e andiamo avanti!

LVALUEs, RVALUEs ed universal references

Eh sì.. ancora una volta ci tocca parlare di *lvalues* e di *rvalues* perchè con il *perfect forwarding* c'entrano, c'entrano eccome.
Per un ripasso di cosa sono vi rimando al mio articolo introduttivo sulla move semantic.
Dicevamo, nella definizione di *perfect forwarding* che le funzioni *target* ricevono gli argomenti con le stesse *proprietà* che avevano quando sono stati passati dal chiamante alla funzione *forwarding.*
Ma quali sono queste proprietà?
Semplice, *sono* che, al momento della chiamata alla funzione *forwarding,* questi argomenti *denotino lvalues* oppure *rvalues*.
Potete tranquillamente pensare a queste proprietà come a *metainformazioni*, che nulla hanno a che spartire con il *valore* trasportato dall'argomento.
Come ormai ben sappiamo, un compilatore C++11 sa riconoscere se un argomento passato ad una funzione denota un *lvalue* oppure un *rvalue* ed è pertanto in grado di chiamare una eventuale funzione *overloaded* corrispondente.
Un esempio notevole è il costruttore di un tipo.
Se il tipo definisce sia costruttore di *copia* che di *move*, il compilatore è in grado di chiamare il costruttore di *move* se riesce a capire che l'argomento passato denota un *rvalue*.
Se il compilatore, per un qualche motivo, non è in grado di inferire questa informazione, allora sarà costretto ad usare il caro vecchio costruttore di *copia*, sacrificando

l’efficienza di una *move*.

universal references

Diamo un’occhiata alla dichiarazione del metodo emplace_back di std::vector:

```
//from the C++11 Standard template <class T, class Allocator = allocator<T>> class vector { public: .. template <class.. Args>
void emplace_back(Args&&... args); .. };
```

emplace_back è un metodo *template* sopra un numero variabile di tipi (*Args*).
Si dicono: *universal reference,* i parametri (come nel caso della emplace_back), o anche il singolo parametro, delle funzioni *template* che hanno la seguente forma: T&& x oppure T&&... x.
Attenzione, per essere una *universal reference*, occorre necessariamente che il parametro sia inserito in una *funzione template*!
Il parametro x di void push_back(T&& x); *non* è una *universal reference*!
push_back è un metodo “*normale*” non un *metodo template*.
Poichè le *universal references* sono *references*, in C++ è obbligatorio che debbano essere inizializzate.
Se l’inizializzazione avviene con un *lvalue,* allora l’*universal reference* corrisponderà ad una *lvalue* reference.
Altrimenti, se l’inizializzazione avviene con un *rvalue,* l’*universal reference* corrisponderà ad una *rvalue* reference.

STD::FORWARD

A questo punto dovremmo aver capito che le *universal reference* in qualche modo riescono a ricordarsi della loro inizializzazione.
È cioè possibile “*interrogarle*” per sapere se la loro *inizializzazione* è avvenuta con un *lvalue* oppure un *rvalue*.
Vi propongo adesso un piccolo test per farvi riflettere se

avete capito o meno dove si vuole andare a parare:

```
//a simple struct, It defines the empty ctor, the copy ctor and the move ctor.
struct MoveMe{ MoveMe(){} MoveMe(const MoveMe &x){ std::cout << "you copied me.. :(" << std::endl; } MoveMe(MoveMe &&x){ std::cout << "congratulation, you actually moved me!" << std::endl; }
}; int main(){ MoveMe IWouldLikeToBeMoved; /*
std::move is necessary because it is not allowed to bind a &&ref from an lvalue.
*/ MoveMe &&ref = std::move(IWouldLikeToBeMoved); MoveMe MoveToMe(ref);
}
```

Quale pensate sia l'output di questo programma?
Pensateci un attimo.
Vi siete convinti che il programma stampi: "*congratulation, you actually moved me!*" ?
Spiacente ragazzi, se è così, allora dovete rivedere le vostre convinzioni su cosa rappresenta *davvero* un *rvalue*.
&&refè una *reference* che può essere inizializzata **solo** da un *rvalue*. Ecco perchè la std::move è necessaria.
Ma una volta inizializzata, ref è a tutti gli effetti un *lvalue*.
La cosa vi stupisce? Allora perchè è possibile ottenere l'indirizzo di ref mediante &ref ?
È cosa nota che da un *lvalue* si possa ricavare l'indirizzo ma *mai* da un *rvalue*.
Ecco perchè era necessario inventare qualcosa come le *universal reference* per implementare il *perfect forwarding:* serviva qualcosa che alla bisogna, potesse dire se l'inizializzazione della reference era avvenuta con un *lvalue* oppure un *rvalue*.
Con una *universal reference* è possibile applicare la funzione std::forward<T>che è in grado di applicare un *cast condizionale* al suo argomento.
Se, l'argomento, originariamente denotava una *lvalue* allora std::forward<T> restituisce una reference *lvalue*.

Se invece, l'argomento originariamente denotava una *rvalue* allora std::forward<T> restituisce una reference *rvalue*.
È quindi certo che l'implementazione di emplace_back internamente, da qualche parte, abbia una chiamata del genere:

```
//from the C++11 Standard template <class T, class Allocator = allocator<T>> class vector { public:
.. template <class.. Args> void emplace_back(Args&&... args){ [..] T(std::forward<decltype(args)>(args ...)); //calls the target T constructor with perfect-forwarded arguments. [..] } .. };
```

Quindi, i costruttori di T saranno chiamati da emplace_back esattamente allo stesso modo come se li avessimo chiamati noi direttamente.
Questa signori, è la magia del *perfect forwarding* dei parametri.

CONCLUSIONI

Bene, spero che questo articolo sia stato di vostro interesse e che vi abbia trasmesso qualcosa che non conoscevate o che non avevate approfondito.
All'atto pratico, nell'uso quotidiano della Standard Template Library, posso sicuramente raccomandare di usare, quando possibile, le funzioni emplace al posto delle classiche funzioni std98.
Molti dei concetti espressi in questo articolo hanno come origine il testo di Scott Meyers: Effective Modern C++.
Per chi volesse approfondire questo ed altri argomenti ne raccomando sicuramente la lettura; senza ombra di dubbio il lavoro di Meyers rappresenta una delle Bibbie del moderno C++.
Grazie a tutti e alla prossima!

https://italiancoders.it/c-perfect-forwarding/