O'REILLY®

Programmazione C++ moderna

42 MODI PER SFRUTTARE AL MEGLIO LE NUOVE FUNZIONALITÀ DI C++11 E C++14

HOEPLI

Scott Meyers

Programmazione C++ moderna

A Darla,
straordinario esemplare nero
di Labrador Retriever

Scott Meyers

Programmazione C++ moderna

42 modi per sfruttare al meglio le nuove funzionalità di C++11 e C++14

EDITORE ULRICO HOEPLI MILANO

Titolo originale: *Effective Modern C++*
Copyright © 2015 Scott Meyers. All rights reserved.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Per l'edizione italiana
Copyright © Ulrico Hoepli Editore S.p.A. 2015
via Hoepli 5, 20121 Milano (Italy) tel. +39 02 864871 – fax +39 02 8052886
e-mail hoepli@hoepli.it

Seguici su Twitter: @Hoepli_1870

www.hoepli.it

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

ISBN EBOOK 978-88-203-6982-8

Traduzione: Paolo Poli
Progetto editoriale: Maurizio Vedovati – Servizi editoriali (info@iltrio.it)
Copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Sommario

Ringraziamenti
Introduzione

1 Deduzione del tipo

Elemento 1 – La deduzione del tipo nei template
  Caso 1 – ParamType è un riferimento o un puntatore, ma non un riferimento universale
  Caso 2 – ParamType è un riferimento universale
  Caso 3 – ParamType non è né un puntatore né un riferimento
  Argomenti array
  Argomenti funzione
Elemento 2 – La deduzione del tipo auto
Elemento 3 – Come funziona decltype
Elemento 4 – Come visualizzare i tipi dedotti
  Editor IDE
  Diagnostiche di compilazione
  Output runtime

2 Il modificatore auto

Elemento 5 – Preferire auto alle dichiarazioni esplicite del tipo
Elemento 6 – Uso di un inizializzatore di tipo esplicito quando auto deduce tipi indesiderati

3 Passare al C++ moderno

Elemento 7 – Distinguere fra () e {} nella creazione degli oggetti
Elemento 8 – Preferire nullptr a 0 e NULL
Elemento 9 – Preferire le dichiarazioni alias ai typedef
Elemento 10 – Preferire gli enum con visibilità a quelli senza visibilità
Elemento 11 – Preferire le funzioni "cancellate" a quelle private undefined
Elemento 12 – Dichiarare con override le funzioni che lo richiedono
Elemento 13 – Preferire i const_iterator agli iterator
Elemento 14 – Dichiarare noexcept le funzioni che non emettono eccezioni
Elemento 15 – Usare constexpr quando possibile
Elemento 16 – Rendere sicure per i thread le funzioni membro const
Elemento 17 – La generazione di funzioni membro speciali

4 I puntatori smart

Elemento 18 – Uso di std::unique_ptr per la gestione delle risorse a proprietà esclusiva
Elemento 19 – Usare std::shared_ptr per la gestione delle risorse a proprietà condivisa
Elemento 20 – Usare std::weak_ptr per puntatori di tipo std::shared_ptr che possono "pendere"
Elemento 21 – Usare std::make_unique e std::make_shared per gestire l'uso di new
Elemento 22 – Idioma Pimpl: definire speciali funzioni membro nel file di implementazione

5 Riferimenti rvalue, semantica di spostamento e perfect-forward

Elemento 23 – Parliamo di std::move e std::forward
Elemento 24 – Distinguere i riferimenti universali e riferimenti rvalue
Elemento 25 – Usare std::move sui riferimenti rvalue e std::forward sui riferimenti universali
Elemento 26 – Evitare l'overloading sui riferimenti universali
Elemento 27 – Alternative all'overloading per riferimenti universali
Abbandonare l'overloading
Passaggio per const T&

Passaggio per valore
L'approccio tag dispatch
Vincolare i template che accettano riferimenti universali
Compromessi
Elemento 28 – Il collasso dei riferimenti
Elemento 29 – Operazioni di spostamento: se non sono disponibili, economiche o utilizzate
Elemento 30 – Casi problematici di perfect-forward
Inizializzatori a graffe
0 o Null come puntatori nulli
Dati membro static const interi solo nella dichiarazione
Nomi di funzioni overloaded e nomi template
Campi bit
Conclusioni

6 Le espressioni lambda
Elemento 31 – Evitare le modalità di cattura di default
Elemento 32 – Usare la cattura iniziale per spostare gli oggetti nelle closure
Elemento 33 – Usate decltype sui parametri auto&& per inoltrarli con std::forward
Elemento 34 – Preferite le lambda a std::bind

7 L'API per la concorrenza
Elemento 35 – Preferire la programmazione basata a task piuttosto che basata a thread
Elemento 36 – Specificare std::launch::async quando è essenziale l'asincronicità
Elemento 37 – Rendere gli std::thread non joinable su tutti i percorsi
Elemento 38 – Attenzione al comportamento variabile del distruttore dell'handle del thread
Elemento 39 – Considerate l'uso di future void per la comunicazione di eventi "one-shot"
Elemento 40 – Usare std::atomic per la concorrenza e volatile per la memoria speciale

8 Tecniche varie
Elemento 41 – Considerare il passaggio per valore per i parametri la cui copia è “poco costosa” e che pertanto vengono sempre copiati
Elemento 42 – Impiegare l’emplacement al posto dell’inserimento

Indice analitico
Informazioni sul Libro
Circa l’autore

Ringraziamenti

Ho iniziato a studiare ciò che allora era chiamato C++0x (il nascente C++11) nel 2009. Ho postato numerose domande sul newsgroup comp.std.c++ e sono grato ai membri di tale comunità (e in particolare a Daniel Krügler) per la grande utilità delle loro risposte. Negli anni più recenti, mi sono rivolto a “Stack Overflow” quando ho avuto domande sul C++11 e il C++14 e sono ugualmente in debito con tale comunità per l’aiuto ricevuto nella comprensione dei dettagli più reconditi della moderna programmazione C++.

Nel 2010 ho preparato il materiale per un corso sul C++0x (pubblicati poi con il nome di *Overview of the New C++*, Artima Publishing, 2010). Questo materiale e la mia conoscenza si sono enormemente avvantaggiati dalla supervisione tecnica svolta da Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober e Anthony Williams. Senza il loro aiuto, probabilmente non sarei stato in grado di scrivere questo libro. Il suo titolo, a proposito, è stato suggerito o comunque sostenuto da numerosi lettori che hanno risposto al mio post *Help me name my book* del 18 febbraio 2014 sul mio blog e Andrei Alexandrescu (autore di *Modern C++ Design*, Addison-Wesley, 2001) ha avuto la gentilezza di benedire la scelta di quel titolo, sostenendo che non attingeva alla sua scelta terminologica.

Mi è davvero impossibile risalire all’origine di tutte le informazioni contenute in questo libro, ma alcune fonti hanno avuto un impatto diretto. Nell’Elemento 4, l’uso di un template indefinito per estrarre le informazioni sul tipo da parte dei compilatori è stato suggerito da Stephan T. Lavavej, e Matt P. Dziubinski ha portato alla mia attenzione Boost. TypeIndex. Nell’Elemento 5, l’esempio unsigned-std::vector<int>::size_type deriva da un articolo del 28 febbraio

2010 di Andrey Karpov: *In what way can C++0x standard help you eliminate 64-bit errors*. L'esempio std::pair<std::string, int>/std::pair<const std::string, int>, sempre in quell'Elemento, è dovuto a una conversazione di Stephan T. Lavavej su "Going Native 2012", *STL11: Magic && Secrets*. L'Elemento 6 è stato ispirato dall'articolo del 12 agosto 2013 di Herb Sutter, *GotW #94 Solution: AAA Style (Almost Always Auto)*. L'Elemento 9 è stato motivato dal post del 27 maggio 2012 sul blog di Martinho Fernandes, *Handling dependent names*. L'esempio dell'Elemento 12 che illustra l'overloading sui qualificatori per riferimenti si basa sulla risposta di Casey alla domanda "What's a use case for overloading member functions on reference qualifiers?", in un post su "Stack Overflow" del 14 gennaio 2014. La mia trattazione nell'Elemento 15 sul supporto espanso del C++14 per le funzioni `constexpr` incorpora informazioni che ho ricevuto da Rein Halbersma. L'Elemento 16 si basa sulla presentazione *C++ and Beyond 2012* di Herb Sutter: "You don't know const and mutable". Il consiglio dell'Elemento 18 di far sì che le funzioni factory restituiscano `std::unique_ptr` si basa sull'articolo *GotW# 90 Solution: Factories* di Herb Sutter del 30 maggio 2013. Nell'elemento 19, `fastLoadWidget` deriva dalla presentazione a "Going Native 2013" di Herb Sutter, *My Favorite C++10-Liner*. La mia trattazione su `std::unique_ptr` e i tipi incompleti dell'Elemento 22 si basa sull'articolo del 27 novembre 2011 di Herb Sutter, *GotW #100: Compilation Firewalls* e anche sulla risposta del 22 maggio 2011 di Howard Hinnant alla domanda su "Stack Overflow": "Is std::unique_ptr<T> required to know the full definition of T?". L'esempio di somma di `Matrix` dell'Elemento 25 si basa sugli scritti di David Abrahams. Il commento dell'8 dicembre 2012 di Joe Argonne al post del 30 novembre 2012 del mio blog, *Another alternative to lambda move capture*, è stato all'origine dell'approccio basato su `std::bind` all'emulazione della cattura iniziale nel C++11, presentata nell'Elemento 32. La spiegazione dell'Elemento 37 del problema con un *detach* implicito nel distruttore di `std::thread` è tratta da un lavoro di Hans-J. Boehm del 4 dicembre 2008, *N2802: A plea to reconsider detach-on-destruction for thread objects*. L'Elemento 41 trae origine dalle discussioni del post di David Abrahams del 15 agosto 2009, dal titolo *Want Speed? Pass by value*. L'idea che i tipi move-only meritino un trattamento speciale è dovuta a Matthew Fioravante, mentre l'analisi della copia basata su assegnamenti deriva dai commenti di Howard Hinnant. Nell'elemento 42, Stephan T. Lavavej e Howard Hinnant mi hanno aiutato a comprendere i profili prestazionali delle funzioni di emplacement e di inserimento e Michael

Winterberg ha richiamato la mia attenzione su come l’emplacement possa portare a uno spreco di risorse (Michael tiene a riconoscere la presentazione a “Going Native 2013” di Sean Parent, *C++ Seasoning* quale sua fonte). Michael ha anche rilevato come le funzioni di emplacement utilizzino l’inizializzazione diretta, mentre le funzioni di inserimento usino l’inizializzazione per copia.

La revisione delle bozze di un testo tecnico è un compito impegnativo, lungo e critico e sono fortunato che così tante persone abbiano accettato di farlo per me. Le bozze, complete o parziali, dell’edizione originale di questo libro sono state ufficialmente rivedute da Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin “:-)” Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews e Tomasz KamiĐski. Ho inoltre ricevuto un feedback da numerosi lettori attraverso O’Reilly’s Early Release EBooks e Safari Books Online’s Rough Cuts, tramite commenti sul mio blog (“The View from Aristeia”) e per posta elettronica. Sono grato a ognuna di queste persone. Questo libro è molto migliore per il fatto di averlo potuto scrivere con il vostro aiuto. Sono in debito, in particolare, con Stephan T. Lavavej e Rob Stewart, le cui annotazioni straordinariamente dettagliate e complete mi hanno portato a pensare che essi abbiano dedicato a questo libro quasi lo stesso tempo che gli ho dedicato io. Ringraziamenti speciali anche a Leor Zolman, che, oltre a rivedere il manoscritto, ha ricontrollato ogni singolo esempio di codice.

Le revisioni dedicate delle versioni digitali del libro sono state eseguite da Gerhard Kreuzer, Emyr Williams e Bradley E. Needham.

La mia decisione di limitare la lunghezza delle righe di codice [nel testo inglese, N.d.T.] a 64 caratteri (il massimo visualizzabile adeguatamente su una pagina stampata e su vari dispositivi digitali, con vari orientamenti di schermo e configurazioni di caratteri) si basa sui dati forniti da Michael Maher.

Ashley Morgan Williams ha reso i miei pasti al ristorante Lake Oswego Pizzicato davvero indimenticabili. Le sue Caesar Salad sono insuperabili.

Dopo più di vent’anni di coabitazione e nonostante la mia attività di autore, mia moglie, Nancy L. Urbano, mi sopporta ancora e ha tollerato i molti mesi di conversazioni distratte con un misto di rassegnazione, esasperazione e lampi di

comprensione e supporto. In questo stesso periodo, il nostro cane, Darla, è stato felice di sonnecchiare durante tutte le ore che ho dedicato a fissare lo schermo del computer, aiutandomi a non dimenticare mai che esiste una vita oltre la tastiera.

Introduzione

Se siete esperti programmatori C++ e mi assomigliate almeno un po', vi siete probabilmente avvicinati al C++11 pensando: "Sì, d'accordo, è sempre il C++, con qualcosina in più". Ma approfondendo l'argomento siete rimasti colpiti dell'entità dei cambiamenti. Le dichiarazioni auto, i cicli for basati su intervalli, le espressioni lambda e i riferimenti rvalue cambiano il volto del C++, per non parlare delle funzionalità dedicate alla concorrenza. E poi ci sono i cambiamenti apportati. 0 e i typedef sono "out", i nullptr e le dichiarazioni alias sono "in". Le enumerazioni devono ora avere campo di visibilità. I puntatori smart sono ora preferibili a quelli standard. Lo spostamento di oggetti è normalmente migliore rispetto alla copia.

Quindi c'è molto da imparare sul C++11, per non parlare del C++14.

Ma soprattutto c'è molto da imparare sull'utilizzo efficace delle nuove funzionalità. Se avete bisogno di informazioni di base sulle funzionalità del C++ "moderno", avete a disposizione molte risorse, ma se cercate una guida sul modo in cui impiegare le funzionalità per creare software corretto, efficiente, di facile manutenzione e portabile, la ricerca è più complicata. Questo è invece esattamente lo scopo di questo libro. È dedicato non a descrivere le funzionalità del C++11 e del C++14, ma piuttosto alla loro applicazione efficace.

Le informazioni di questo libro sono suddivise in linee guida chiamate Elementi. Volete scoprire le varie forme della deduzione del tipo? O sapere quando usare le dichiarazioni `auto` e quando no? Siete interessati ai motivi per cui le funzioni membro `const` debbano essere sicure rispetto ai thread, a come implementare l'Idioma Pimpl utilizzando `std::unique_ptr`, a perché si dovrebbe evitare la modalità di cattura di default nelle espressioni lambda o alle differenze fra

`std::atomic` e `volatile`? Qui troverete tutte queste risposte. Il tutto in un modo indipendente dalla piattaforma: risposte conformi allo standard. Questo libro parla di un C++ *portabile.*

Gli Elementi di questo libro sono semplici indicazioni, non regole, poiché le indicazioni prevedono eccezioni. La parte più importante di ciascun Elemento non è il consiglio che offre, ma il ragionamento su cui si basa tale consiglio. Dopo averlo letto, potrete determinare se le specifiche circostanze del vostro progetto giustificano una violazione delle indicazioni fornite nell'Elemento. Il vero obiettivo di questo libro non è quello di dirvi cosa fare e cosa evitare, ma di offrirvi una conoscenza più profonda del modo in cui funzionano le cose in C++11 e C++14.

Terminologia e convenzioni

Per essere sicuri di comprendersi, è importante accordarsi su una certa terminologia, partendo, ironicamente, dal termine "C++". Vi sono state quattro versioni ufficiali del C++, ognuna delle quali denominata in base all'anno in cui è stato adottato il corrispondente Standard ISO: C++98, C++03, C++11 e C++14. Il C++98 e il C++03 differiscono solo per dettagli tecnici e pertanto, in questo libro, ci riferiamo a entrambi con il nome di C++98. Quando parlo di C++11, intendo sia il C++11 sia il C++14, in quanto quest'ultimo è sostanzialmente un'estensione del primo. Quando invece parlo di C++14, intendo esclusivamente il C++14. Infine, quando parlo di C++, sto facendo un'affermazione generale, che riguarda tutte le versioni del linguaggio.

Termine utilizzato	Versioni del linguaggio corrispondente
C++	Tutte
C++98	C++98 e C++03
C++11	C++11 e C++14
C++14	C++14

Pertanto, potrei dire che il C++ pone l'accento sull'efficienza (un'affermazione vera per tutte le versioni), che il C++98 non offre il supporto per la concorrenza

(vero per il C++98 e il C++03), che il C++11 supporta le espressioni lambda (vero per il C++11 e il C++14) e che il C++14 offre la deduzione generalizzata del tipo restituito dalle funzioni (vero solo per il C++14).

La funzionalità più pervasiva del C++11 è probabilmente la semantica degli spostamenti e alla base della semantica degli spostamenti vi è il fatto di distinguere le espressioni che sono rvalue da quelle che sono lvalue. Questo perché gli rvalue indicano oggetti impiegabili per operazioni di spostamento, mentre gli lvalue, generalmente, no. In teoria (anche se non sempre in pratica), gli rvalue corrispondono a oggetti temporanei restituiti dalle funzioni, mentre gli lvalue corrispondono a oggetti cui si può fare riferimento, per nome oppure seguendo un puntatore o un riferimento lvalue.

Un'utile euristica per determinare se un'espressione è un lvalue consiste nel chiedere se è possibile prenderne l'indirizzo. Se è possibile, si tratta di una lvalue. Se non è possibile, normalmente si tratta di un rvalue. Un'ottima caratteristica di questa euristica è il fatto che aiuta a ricordare che il tipo di un'espressione è indipendente dal fatto che l'espressione sia un lvalue o un rvalue. In pratica, dato un tipo `T`, è possibile avere lvalue di tipo `T` e anche rvalue di tipo `T`. È particolarmente importante ricordarlo quando si ha a che fare con un parametro di un tipo di riferimento rvalue, poiché il parametro stesso è un lvalue:

```
class Widget {
public:
   Widget(Widget&& rhs);          // rhs è un lvalue, anche se il suo tipo
   …                              // è riferimento rvalue
};
```

Qui, è perfettamente valido prendere l'indirizzo di `rhs` all'interno del costruttore per spostamento di `Widget`, cosicché `rhs` è un lvalue anche se il suo tipo è un riferimento rvalue (ragionando in modo analogo, tutti i parametri sono lvalue).

Questo frammento di codice illustra varie convenzioni adottate nel testo.

- Il nome della classe è `Widget`. Utilizzo il termine `Widget` ogni volta che voglio fare riferimento a un arbitrario tipo definito dall'utente. A meno che debba mostrare dettagli specifici della classe, utilizzo `Widget` senza dichiararlo.

- Utilizzo il nome di parametro *rhs (“right-hand side”)*. È il nome di parametro che uso di solito per le *operazioni di spostamento* (ovvero per il costruttore per spostamento e per l’operatore di assegnamento per spostamento) e per le *operazioni di copia* (ovvero per il costruttore per copia e per l’operatore di assegnamento per copia). Lo impiego anche per il parametro destro degli operatori binari:

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  Nessuna sorpresa, spero, che *lhs* significhi *“left-hand side”*.

- Applico una particolare formattazione a quelle parti del codice o dei commenti sui quali intendo attrarre la vostra attenzione. Nel costruttore per spostamento di `Widget` precedente, ho evidenziato la dichiarazione di `rhs` e la parte del commento che nota che `rhs` è un lvalue. Il codice evidenziato non è necessariamente buono o cattivo. È codice cui occorre dedicare particolare attenzione.

- Utilizzo il simbolo “…” per indicare “qui occorre inserire del codice”. Questi puntini stretti (…) sono diversi da quelli larghi (`...`) che ho utilizzato nel codice sorgente per i template del C++11. La cosa, spiegata così all’inizio del libro, può sembrare confusa, ma non lo è. Ecco un esempio:

```
template<typename... Ts>                   // questi sono i puntini
void processVals(const Ts&... params)      // del linguaggio C++
{
   …                                       // questi, invece, significano
                                           // “qui ci va del codice”
}
```

  La dichiarazione di `processVals` mostra che utilizzo `typename` per dichiarare i tipi di parametri nei template, ma questa è solo una preferenza personale; la parola chiave `class` avrebbe funzionato altrettanto bene. In quei casi in cui mostro estratti di codice da uno standard del C++, dichiaro i parametri di tipo utilizzando `class`, poiché questo è ciò che fa anche lo Standard.

Quando un oggetto viene inizializzato con un altro oggetto dello stesso tipo, il

nuovo oggetto si dice che è una *copia* dell'oggetto che esegue l'inizializzazione, anche se la copia è stata creata tramite un costruttore per spostamento. Purtroppo non esiste in C++ una terminologia che distingua un oggetto che è una copia costruita per copia da un oggetto che è una copia costruita per spostamento:

```
void someFunc(Widget w);          // il parametro w di someFunc
                                  // viene passato per valore

Widget wid;                       // wid è un Widget

someFunc(wid);                    // in questa chiamata a someFunc,
                                  // w è una copia di wid creata
                                  // tramite costruzione per copia

someFunc(std::move(wid));         // in questa chiamata a SomeFunc,
                                  // w è una copia di wid creata
                                  // tramite costruzione per
                                  spostamento
```

Le copie di rvalue sono generalmente costruite per spostamento, mentre le copie di lvalue sono normalmente costruite per copia. Un'implicazione è che se sappiamo solamente che un oggetto è una copia di un altro oggetto, non è possibile dire quanto sia costoso eseguirne la copia. Nel codice precedente, per esempio, non vi è alcun modo per dire quanto possa essere costoso creare il parametro `w` senza sapere se a `someFunc` sono stati passati rvalue o lvalue (bisognerebbe anche conoscere il costo che comporta spostare e copiare dei `Widget`).

In una chiamata a funzione, le espressioni passate nel punto di chiamata sono gli *argomenti* della funzione. Gli argomenti vengono utilizzati per inizializzare i parametri della funzione. Nella prima chiamata alla `someFunc` precedente, l'argomento è `wid`. Nella seconda chiamata, l'argomento è `std::move(wid)`. In entrambe le chiamate, il parametro è `w`. La distinzione fra argomenti e parametri è importante, poiché i parametri sono rvalue, mentre gli argomenti con cui vengono inizializzati possono essere rvalue o lvalue. Questo è rilevante in particolare durante il processo di *perfect-forward*, nel quale un argomento passato a una funzione viene passato a una seconda funzione, in modo che venga

conservato il fatto che l’argomento originale era un rvalue oppure un lvalue (si parla in dettaglio di perfect-forward nell’Elemento 30).

Le funzioni ben realizzate sono *sicure rispetto alle eccezioni,* ovvero offrono almeno una *garanzia di sicurezza di base* rispetto alle eccezioni. Tali funzioni garantiscono al chiamante che anche se viene lanciata un’eccezione, gli elementi invarianti del programma rimarranno intatti (ovvero le strutture dati non verranno danneggiate) e non vi sarà alcuno spreco di risorse. Le funzioni che offrono una *garanzia di sicurezza forte* rispetto alle eccezioni garantiscono ai chiamanti che se viene lanciata un’eccezione, lo stato del programma rimane quello precedente alla chiamata.

Quando parlo di *oggetto funzione*, normalmente intendo un oggetto di un tipo che supporta una funzione membro operator(). In altre parole, un oggetto che si comporta come una funzione. Occasionalmente uso il termine in un senso un po’ più generale, per indicare qualsiasi cosa che possa essere richiamata utilizzando la sintassi di una chiamata a funzione non membro (ovvero `nomeFunzione(argomenti))`. Questa definizione più ampia copre non solo gli oggetti che supportano operator(), ma anche le funzioni e i puntatori a funzione in stile C (la definizione più ristretta proviene dal C++98, mentre quella più ampia deriva dal C++11). Generalizzando ulteriormente, con l’aggiunta dei puntatori a funzione membro, si ottengono gli *oggetti richiamabili*. Generalmente è possibile ignorare questa distinzione e considerare semplicemente gli oggetti funzione e gli oggetti richiamabili come oggetti C++ che possono essere richiamati utilizzando una specifica sintassi di chiamata a funzione.

Gli oggetti funzione creati attraverso espressioni lambda sono chiamati *closure.* È raro che sia necessario distinguere fra espressioni lambda e le *closure* che esse creano, pertanto spesso faccio riferimento a entrambe le cose con il nome di *lambda.* Analogamente, raramente distinguo fra *template di funzioni* (ovvero template che generano funzioni) *e funzioni template* (ovvero le funzioni generate partendo dai template di funzioni). La stessa cosa vale per i *template di classi* e le *classi template.*

Molti elementi del C++ possono essere sia dichiarati sia definiti. Le *dichiarazioni* introducono i nomi e i tipi senza fornire dettagli, ovvero dove sarà situata la loro area di memoria e come verranno implementati questi elementi:

```
extern int x;                    // dichiarazione di oggetto

class Widget;                    // dichiarazione di classe

bool func(const Widget& w);      // dichiarazione di funzione

                                 // dichiarazione di enum
enum class Color;                // con campo di visibilità (vedi
                                 Elemento 10)
```

Le *definizioni* forniscono indicazioni sulla zona di memoria o i dettagli implementativi:

```
int x;                           // definizione di oggetto

class Widget {                   // definizione di classe
…
};

bool func(const Widget& w)
{ return w.size() < 10; }        // definizione di funzione

enum class Color
{ Yellow, Red, Blue };           // definizione di enum con campo
                                 di visibilità
```

Una definizione funge anche da dichiarazione e, pertanto, a meno che sia davvero importante che una certa cosa sia una definizione, tendo far a far riferimento alle dichiarazioni.

Definisco la *signature* di una funzione come quella parte della dichiarazione che specifica il tipo dei parametri e del valore restituito. I nomi della funzione e dei parametri non fanno parte della signature. Nell'esempio precedente, la signature di `func` è `bool (const Widget&)`. Gli elementi della dichiarazione di una funzione che non sono il tipo dei parametri e il tipo che restituisce (ovvero `noexcept` e `constexpr`, se presenti) sono esclusi. (`noexcept` e `constexpr` sono descritti negli Elementi 14 e 15). La definizione ufficiale di "signature" è leggermente differente dalla mia ma, per quanto riguarda questo libro, la mia definizione è più utile (la definizione ufficiale omette, per certi versi, il tipo

restituito).

I nuovi Standard del C++ generalmente preservano la validità del codice scritto per gli Standard precedenti ma, occasionalmente, il Comitato di Standardizzazione specifica funzionalità deprecate. Tali funzionalità sono nella "lista nera" degli Standard e potrebbero essere rimosse dalle prossime versioni. I compilatori possono avvisarvi o meno del fatto che utilizzate funzionalità deprecate, ma fareste meglio a evitarle. Non solo potreste avere problemi futuri di portabilità ma, generalmente, esistono funzionalità migliori che le sostituiscono. Per esempio, `std::auto_ptr` è deprecato in C++11, poiché `std::unique_ptr` svolge lo stesso lavoro, ma molto meglio.

Talvolta uno Standard dice che il risultato di un'operazione è un *comportamento indefinito*. Ciò significa che il comportamento runtime è imprevedibile ed è inutile dire che bisogna tenersi alla larga da ogni incertezza. Gli esempi di azioni con comportamento indefinito comprendono l'uso delle parentesi quadre (`[ ]`) con indici che vanno oltre i limiti di uno `std:vector`, il deindirizzamento di un iteratore non inizializzato o la creazione di una competizione sui dati (ovvero due o più thread, di cui almeno uno scrive, i quali accedono simultaneamente alla stessa area di memoria).

I vecchi puntatori del C++, come quelli restituiti da `new`, sono chiamati nel libro *puntatori standard*. L'opposto di un puntatore standard è un *puntatore smart*. I puntatori smart, normalmente, eseguono l'overload degli operatori di deindirizzamento dei puntatori (`operator->` e `operator*`) anche se l'Elemento 20 spiega che `std::weak_ptr` rappresenta un'eccezione.

Nei commenti del codice sorgente, talvolta abbrevio con i termini `costr.` e `distr.` le parole "costruttore" e "distruttore".

1

Deduzione del tipo

Il C++98 ha un unico set di regole per la deduzione del tipo: quello dei template delle funzioni. Il C++11 modifica un po' questo insieme di regole aggiungendone altre due, una per `auto` e un'altra per `decltype`. Il C++14, da parte sua, estende i contesti d'uso in cui possono essere impiegati `auto` e `decltype`. L'impiego sempre più esteso della deduzione del tipo ci libera dall'obbligo di specificare sempre quei tipi che sono ovvi o ridondanti. Ciò rende il software C++ più adattabile, in quanto basta modificare un tipo in un punto del codice sorgente; grazie alla deduzione del tipo, tale modifica si propagherà automaticamente anche in altri punti. Tuttavia, ciò può anche complicare l'interpretazione del codice, in quanto i tipi dedotti dai compilatori potrebbero non essere così ovvi come sembrano.

Senza una solida conoscenza del funzionamento della deduzione del tipo, sarebbe impossibile programmare in modo efficace in C++ moderno. Semplicemente vi sono troppi contesti in cui "scatta" la deduzione del tipo: nelle chiamate ai template di funzioni, nella maggior parte delle situazioni in cui compare auto, nell'espressione decltype e, in C++14, dove viene impiegato l'enigmatico costrutto decltype(auto).

Questo capitolo fornisce informazioni relative alla deduzione del tipo che ogni sviluppatore C++ dovrebbe conoscere. Spiega come funziona la deduzione del tipo nei template, come si può realizzare con `auto` e come funziona decltype.

Spiega anche come si possono costringere i compilatori a rivelare quali tipi hanno dedotto, rassicurandoci sul fatto che i compilatori stiano effettivamente deducendo i tipi che noi desideriamo.

Elemento 1 – La deduzione del tipo nei template

Quando accade che chi usa un sistema complesso, in realtà non sa esattamente come funziona, ma ne è soddisfatto, questo la dice lunga sull'efficacia del sistema stesso. In questo senso, la deduzione del tipo nei template in C++ è un grande successo. Milioni di programmatori passano argomenti a funzioni template con risultati pienamente soddisfacenti, anche se molti di questi programmatori sarebbero in difficoltà a descrivere dettagliatamente il modo in cui vengono dedotti i tipi impiegati da tali funzioni.

Se vi ritrovate in questo esempio, per voi c'è una buona notizia e anche una cattiva notizia. La buona notizia è che la deduzione del tipo per i template è alla base di una delle funzionalità più interessanti del C++ moderno: auto. Se vi piace il modo in cui il C++98 deduce i tipi per i template, non avrete problemi con il modo in cui il C++11 deduce i tipi per auto. La cattiva notizia è che quando le regole di deduzione del tipo nei template vengono applicate nel contesto di auto, a volte sono meno intuitive di quando vengono applicate ai template. Per questo motivo, è importante comprendere appieno gli aspetti della deduzione del tipo nei template realizzata da auto. Questo Elemento tratta esattamente questo argomento.

Ragionando in termini di pseudocodice, possiamo considerare il template di una funzione nel seguente modo:

```
template<typename T>
void f(ParamType param);
```

Una chiamata avrà dunque il seguente aspetto:

```
f(expr);                          // richiama f su una certa
                                  espressione
```

Durante la compilazione, i compilatori usano *expr* per dedurre due tipi: per `T` e

per *ParamType*. Normalmente, questi tipi sono differenti, poiché spesso *ParamType* contiene delle aggiunte, come *const* o altri qualificatori per riferimenti. Per esempio, se il template è dichiarato nel seguente modo,

```
template<typename
T>
void f(const T&        // ParamType è
param);                const T&
```

e abbiamo questa chiamata,

```
int x = 0;

f(x);                              // richiama f con un int
```

T viene dedotto come int, ma *ParamType* viene dedotto come const int&.

È ragionevole aspettarsi che il tipo dedotto per T coincida con il tipo di argomento passato alla funzione, ovvero che T sia il tipo di *expr*. Nell'esempio precedente, le cose stanno così: x è un int e T viene dedotto come int. Ma le cose non funzionano sempre così. Il tipo dedotto per T dipende non solo dal tipo di *expr*, ma anche dalla forma di *ParamType*. Vi sono tre casi.

- *ParamType* è un puntatore di tipo riferimento, ma non è un riferimento universale (i riferimenti universali sono descritti nell'Elemento 24. A questo punto, tutto ciò che è necessario sapere è che esistono e che non sono la stessa cosa dei riferimenti lvalue o rvalue).
- *ParamType* è un riferimento universale.
- *ParamType* non è né un puntatore né un riferimento.

Pertanto abbiamo tre scenari di deduzione da esaminare. Ognuno di essi si baserà sulla forma generale del template:

```
template<typename T>
void f(ParamType param);

f(expr);                           // deduce T e ParamType da expr
```

Caso 1 – ParamType è un riferimento o un puntatore, ma non un riferimento universale

La situazione più semplice si ha quando `ParamType` è un riferimento o un puntatore, ma non un riferimento universale. In tal caso, la deduzione del tipo funziona nel seguente modo.

1. Se il tipo di `expr` è un riferimento, ignora la parte del riferimento.
2. Poi esegui un pattern-match di `expr` rispetto al `ParamType`, per determinare `T`.

Per esempio, se il template è il seguente,

```
template<typename
T>
void f(T& param);    // param è un riferimento
```

e abbiamo le seguenti dichiarazioni di variabili,

```
int x = 27;          // x è un int
const int cx = x;    // cx è un const int
const int& rx = x;   // rx è un riferimento a x, const int
```

i tipi dedotti per `param` e `T` nelle varie chiamate sono i seguenti:

```
f(x);                // T è int, param è int&

f(cx);               // T è const int,
                     // param è const int&


f(rx);               // T è const int,
                     // param è const int&
```

Nella seconda e nella terza chiamata, notate che, poiché `cx` e `rx` designano valori `const`, `T` viene dedotto come `const int`, fornendo pertanto un tipo di parametro `const int&`. Questo è importante per il chiamante. Quando il chiamante passa un oggetto `const` a un parametro riferimento, si attende che l'oggetto rimanga non modificabile, ovvero che il parametro sia un riferimento a

const. Questo è il motivo per cui è sicuro passare un oggetto const al template che accetta un parametro T&: la "costanza" dell'oggetto diviene parte del tipo dedotto per T.

Nel terzo esempio, notate che anche se il tipo di rx è un riferimento, T viene dedotto come un non-riferimento. Il fatto che rx sia un riferimento viene ignorato durante la deduzione del tipo.

Questi esempi impiegano tutti dei parametri che sono riferimenti lvalue, ma la deduzione del tipo funziona esattamente allo stesso modo per i parametri riferimenti rvalue. Naturalmente, ai parametri riferimento rvalue possono essere passati solo argomenti rvalue, ma tale restrizione non ha nulla a che fare con la deduzione del tipo.

Se cambiamo il tipo del parametro di f da T& a const T&, le cose cambiano un po', ma non in modo poi così drastico. Il fatto che cx e rx siano costanti continua a essere rispettato, ma, poiché ora supponiamo che param sia un riferimento a const, non vi è più la necessità che const venga dedotto come parte di T:

```
template<typename T>
void f(const T& param);     // ora param un rif-a-const

int x = 27;                 // come prima
const int cx = x;           // come prima
const int& rx = x;          // come prima

f(x);                       // T è int, il tipo di param è const
                            // int&

f(cx);                      // T è int, il tipo di param è const
                            // int&

f(rx);                      // T è int, il tipo di param è const
                            // int&
```

Come prima, il fatto che rx sia un riferimento viene ignorato durante la deduzione del tipo.

Se param fosse stato un puntatore (o un puntatore a const) invece che un riferimento, le cose sarebbero funzionate sostanzialmente allo stesso modo:

```
template<typename
T>
void f(T* param);    // param ora è un puntatore

int x = 27;          // come prima
const int *px = &x;  // px è un puntatore a x come const int

f(&x);               // T è int, il tipo di param è int*

f(px);               // T è const int,
                     // il tipo di param è const int*
```

A questo punto potreste sentirvi quasi annoiati da questi dettagli, poiché le regole di deduzione del tipo del C++ funzionano in modo così naturale per i parametri riferimento e puntatore che leggere questo argomento potrebbe non sembrarvi una grande novità. Tutto è, semplicemente, ovvio! Questo è esattamente ciò che desideriamo da un sistema di deduzione del tipo.

Caso 2 – ParamType è un riferimento universale

Le cose si fanno meno ovvie per i template che accettano come parametro un riferimento universale. Tali parametri sono dichiarati come riferimenti rvalue (ovvero in un template di funzione che accetta un parametro di tipo `T`, il tipo dichiarato per il riferimento universale è `T&&`), ma le cose funzionano in modo differente quando vengono passati argomenti lvalue. Questo fatto viene raccontato per esteso nell'Elemento 24, ma la sostanza è la seguente.

- Se *`expr`* è un lvalue, sia `T` sia *`ParamType`* vengono dedotti come riferimenti lvalue. Questo è doppiamente insolito. Innanzitutto, è l'unica situazione nella deduzione del tipo di un template in cui `T` viene dedotto come riferimento. In secondo luogo, sebbene *`ParamType`* sia dichiarato utilizzando la sintassi per un riferimento rvalue, il tipo dedotto è un riferimento lvalue.
- Se *`expr`* è un rvalue, valgono le regole "normali" (ovvero quelle del Caso 1).

Per esempio:

```
template<typename
T>
void f(T&& param);   // param ora è un riferimento universale

int x = 27;          // come prima
const int cx = x;    // come prima
const int& rx = x;   // come prima

f(x);                // x è un lvalue, pertanto T è int&,
                     // il tipo di param è also int&

f(cx);               // cx è un lvalue, pertanto T è const int&,
                     // il tipo di param è also const int&

f(rx);               // rx è un lvalue, pertanto T è const int&,
                     // anche il tipo di param è const int&

f(27);               // 27 è un rvalue, pertanto T è int,
                     // quindi il tipo di param è int&&
```

L'Elemento 24 spiega esattamente perché questi esempi funzionano in questo modo. Qui, il concetto chiave, è che le regole di deduzione del tipo per i parametri che sono riferimenti universali sono differenti rispetto a quelle per i parametri che sono riferimenti lvalue o rvalue. In particolare, quando vengono impiegati riferimenti universali, la deduzione del tipo distingue fra argomenti lvalue e argomenti rvalue. Ciò non accade mai per i riferimenti non universali.

Caso 3 – ParamType non è né un puntatore né un riferimento

Quando `ParamType` non è né un puntatore né un riferimento, abbiamo a che fare con un passaggio per valore:

```
template<typename
T>
void f(T param);     // param ora viene passato per valore
```

Questo significa che `param` sarà una copia di ciò che gli viene passato, un oggetto completamente nuovo. Il fatto che param sia un nuovo oggetto determina le regole che governano il modo in cui `T` viene dedotto da *`expr`*.

1. Come prima, se il tipo di *`expr`* è un riferimento, si deve ignorare la parte riferimento.
2. Se, dopo aver ignorato il fatto che *`expr`* è un riferimento, *`expr`* è const, occorre ignorare anche questo. Se è volatile, si deve ignorare anche questo (gli oggetti volatile sono poco comuni e generalmente vengono utilizzati solo per l'implementazione dei driver; per informazioni consultate l'Elemento 40).

Pertanto:

```
int x = 27;           // come prima
const int cx = x;     // come prima
const int& rx = x;    // come prima

f(x);                 // i tipi di T e param sono entrambi int

f(cx);                // i tipi di T e param sono ancora entrambi
                      int


f(rx);                // i tipi di T e param sono sempre entrambi
                      int
```

Notate che anche se `cx` e `rx` rappresentano valori `const`, `param` non è `const`. Questo ha perfettamente senso. `param` è un oggetto completamente indipendente da `cx` e `rx`, è una *copia* di `cx` o `rx`. Il fatto che `cx` e `rx` non possano essere modificati non dice nulla sul fatto che lo possa essere anche `param`. Questo è il motivo per cui il fatto che `expr` sia costante (ed, eventualmente, `volatile`) viene ignorato quando si deduce il tipo di `param`: per il solo fatto che `expr` non può essere modificato, questo non significa che una sua copia non debba esserlo.

È importante riconoscere che `const` (e `volatile`) vengono ignorati solo per i parametri passati per valore. Come abbiamo visto, per i parametri passati per riferimento o per puntatore a `const`, la "costanza" di `expr` viene conservata durante la deduzione del tipo. Ma considerate il caso in cui `expr` sia un puntatore `const` a un oggetto `const` ed `expr` venga passata a un `param` per valore:

```
template<typename T>
void f(T param);            // param viene sempre passato per valore

const char* const ptr =     // ptr è puntatore const a un oggetto
                            const
   "Fun with pointers";


f(ptr);                     // passa un argomento di tipo const char
                            * const
```

Qui, il `const` a destra dell'asterisco dichiara `ptr` come `const`: `ptr` non può essere fatto puntare a un altro indirizzo, né può essere impostato a `null` (il `const` a sinistra dell'asterisco dice a cosa punta `ptr`, la stringa di caratteri, è `const`, pertanto non può essere modificata). Quando a `f` viene passato `ptr`, i bit che costituiscono il puntatore vengono copiati in `param`. *Pertanto, il puntatore stesso (`ptr`) viene passato per valore.* In base alla regola di deduzione del tipo per i parametri passati per valore, il fatto che `ptr` sia costante verrà ignorato e il tipo dedotto per `param` sarà `const char*`, ovvero un puntatore modificabile a una stringa di caratteri `const`. Il fatto che ciò a cui punta `ptr` sia costante viene mantenuto durante la deduzione del tipo, ma la "costanza" di `ptr` viene ignorata quando lo si copia per creare un nuovo puntatore, `param`.

Argomenti array

Quanto detto finora riguarda la situazione generale della deduzione del tipo di un template, ma esiste un caso di nicchia di cui vale la pena di parlare. Riguarda il fatto che i tipi array sono differenti dai tipi puntatore, anche se talvolta sembrano poter essere impiegati l'uno al posto dell'altro. Un grande contributo a questa illusione è il fatto che, in molti contesti, un array *decade*, trasformandosi in un puntatore al suo primo elemento. Questo decadimento è ciò che permette la compilazione di codice come il seguente:

```
const char name[] = "J. P.       // il tipo di name è
Briggs";
                                 // const char[13]


const char * ptrToName = name;   // l'array decade in un puntatore
```

Qui il puntatore `ptrToName`, dichiarato come `const char*`, viene inizializzato con `name`, che è `const char[13]`. Questi tipi (`const char*` e `const char[13]`) non sono la stessa cosa, ma a causa della regola del decadimento dell'array in puntatore, questo codice passa la compilazione.

Ma cosa accade se un array viene passato a un template prendendo un parametro per valore? Cosa accade in questo caso?

```
template<typename T>
void f(T param);          // template con un parametro passato per valore
f(name);                  // quali tipi vengono dedotti per T e param?
```

Iniziamo osservando che non esiste una cosa come un parametro di funzione che è un array. Sì, certamente, la seguente sintassi è legale,

```
void myFunc(int param[]);
```

ma la dichiarazione dell'array viene trattata come la dichiarazione di un puntatore, ovvero `myFunc` potrebbe essere dichiarato in modo equivalente così:

```
void myFunc(int* param);   // stessa funzione di prima
```

Questa equivalenza dei parametri array e puntatore "spunta dalle radici" C, che sono alla base del C++, e supporta l'illusione che i tipi array e puntatore siano la stessa cosa.

Poiché le dichiarazioni di parametri array vengono trattate come se fossero parametri puntatore, il tipo di un array che viene passato a una funzione template per valore viene dedotto come un tipo puntatore. Ciò significa che nella chiamata al template `f`, il suo parametro di tipo `T` viene dedotto come `const char*`:

```
f(name);                  // name è un array, ma T viene dedotto come const
char*
```

Ma veniamo al problema: anche se le funzioni non possono dichiarare parametri che siano davvero array, *possono* dichiarare parametri che sono *riferimenti* ad

array! Così, se modifichiamo il template `f` in modo che prenda il proprio argomento per riferimento,

```
template<typename T>
void f(T& param);           // template con parametro per-riferimento
```

e gli passiamo un array,

```
f(name);                    // passaggio di un array a f
```

il tipo dedotto per `T` è, in realtà, un array! Tale tipo include le dimensioni dell'array e dunque, in questo esempio, `T` viene dedotto come `const char [13]` e il tipo del parametro di `f` (un riferimento a questo array) è `const char (&)[13]`. D'accordo, la sintassi è intricata, ma il fatto di saperlo vi darà parecchi "punti", agli occhi di quelle poche persone che considerano questi dettagli.

Una cosa interessante è che la possibilità di dichiarare riferimenti ad array consente di creare un template che deduce il numero di elementi contenuto nell'array:

```
// restituisce le dimensioni di un array come una costante di compilazione.
// (Il parametro array non ha nome, poiché ci interessa solo
// il numero di elementi che contiene.)
template<typename T, std::size_t N>                  // vedere descrizione
constexpr std::size_t arraySize(T (&)[N]) noexcept   // vedere sotto
{                                                    // constexpr
return N;                                            // e
}                                                    // noexcept
```

Come descritto nell'Elemento 15, la dichiarazione di questa funzione `constexpr` rende disponibile il suo risultato durante la compilazione. Ciò consente di dichiarare, per esempio, un array con lo stesso numero di elementi di un secondo array le cui dimensioni vengono calcolate sulla base degli inizializzatori posti fra parentesi graffe:

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };     // keyVals ha
                                                // 7 elementi

int mappedVals[arraySize(keyVals)];             // e anche
                                                // mappedVals
```

Naturalmente, in quanto sviluppatori C++ moderni, preferireste impiegare uno `std::array` rispetto a un array standard:

```
std::array<int, arraySize(keyVals)>             // mappedVals'
mappedVals;
                                                // size è 7
```

Quanto al fatto che `arraySize` venga dichiarato come `noexcept`, questo ha lo scopo di aiutare i compilatori a generare codice migliore. Per i dettagli, consultate l'Elemento 14.

Argomenti funzione

Gli array non sono l'unico elemento del C++ che decade in un puntatore. Anche i tipi funzione possono decadere in puntatore a funzione e tutto ciò di cui abbiamo parlato a proposito della deduzione del tipo per gli array si applica anche alla deduzione del tipo per le funzioni e al loro decadimento in puntatori a funzione. Il risultato è che:

```
void someFunc(int,          // someFunc è una funzione;
double);
                            // il tipo è void(int, double)


template<typename T>
void f1(T param);           // in f1, param passato per valore


template<typename T>
void f2(T& param);          // in f2, param passato per riferimento


f1(someFunc);               // param dedotto come puntatore-a-
                            funzione;
```

```
                                // il tipo è void (*)(int, double)

f2(someFunc);                   // param dedotto come riferimento-a-funzione;
                                // il tipo è void (&)(int, double)
```

Questo raramente produce una reale differenza pratica, ma se dovete conoscere il decadimento da array in puntatore, dovete anche conoscere il decadimento da funzione a puntatore.

Si applicano le regole relative ad `auto` per la deduzione del tipo nei template. Abbiamo indicato all'inizio che le regole sono piuttosto intuitive e, nella maggior parte dei casi, le cose stanno proprio così. Lo speciale trattamento accordato agli lvalue quando si deduce il tipo per i riferimenti universali, però, intorbidisce un po' le acque, e le regole che governano il decadimento in puntatore di array e funzioni complicano ulteriormente le cose. Talvolta viene voglia di prendere per il… collo il compilatore e chiedergli "Dimmi quale razza di tipo stai deducendo!". In questi casi, consultate l'Elemento 4: vedrete come convincere i compilatori a fare quello che volete.

Argomenti da ricordare

- Durante la deduzione del tipo nei template, gli argomenti che sono riferimenti vengono trattati come non riferimenti: il fatto che siano riferimenti viene ignorato.
- Nella deduzione del tipo per i parametri riferimenti universali, gli argomenti lvalue ricevono un trattamento particolare.
- Nella deduzione del tipo per parametri passati per valore, gli argomenti `const` e/o volatile vengono trattati come se fossero non**`-const`** e non**`-volatile`**.
- Nella deduzione del tipo nei template, gli argomenti che sono array o nomi di funzioni decadono in puntatori, a meno che vengano utilizzati per inizializzare i riferimenti.

Elemento 2 – La deduzione del tipo auto

Avendo letto l'Elemento 1 dedicato alla deduzione del tipo nei template, sapete già quasi tutto ciò che occorre sapere sulla deduzione del tipo `auto`, poiché, con un'unica eccezione, la deduzione del tipo `auto` *è* una deduzione di tipo da template. Ma come può essere? La deduzione del tipo nei template riguarda template, funzioni e parametri, mentre `auto` non ha a che fare con nessuna di queste cose.

Questo è vero, ma non è così importante. Esiste uno stretto legame tra deduzione del tipo nei template e deduzione del tipo `auto`. Esiste letteralmente una trasformazione algoritmica dall'uno all'altro.

Nell'Elemento 1, la deduzione del tipo per template viene descritta utilizzando questo template generico di funzione:

```
template<typename T>
void f(ParamType param);
```

e questa chiamata generica:

```
f(expr);                   // richiama f su una certa espressione
```

Nel caso di f, i compilatori usano *`expr`* per dedurre i tipi per `T` e *`ParamType`*.

Quando una variabile viene dichiarata utilizzando `auto`, quest'ultimo gioca il ruolo di `T` nel template e lo specificatore di tipo per la variabile si comporta come *`ParamType`*. È più facile mostrarlo che descriverlo, pertanto considerate il seguente esempio:

`auto` x = 27;

Qui, lo specificatore di tipo per x è semplicemente `auto` e basta. Al contrario, nella seguente dichiarazione,

`const auto` cx = x;

lo specificatore di tipo è `const auto`.

E qui,

```
const auto& rx = x;
```

lo specificatore di tipo è `const auto&`. Per dedurre il tipo di `x`, `cx` e `rx` in questi esempi, i compilatori si comportano come se vi fosse un template per ogni dichiarazione e anche una chiamata a tale template con la relativa espressione di inizializzazione:

```
template<typename T>          // template concettuale per
void func_for x(T param);     // dedurre il tipo di x

func_for_x(27);               // chiamata concettuale:
                              // il tipo dedotto per param è il tipo
                              di x
template<typename T>          // template concettuale per
void func_for_cx(const T
param);                       // dedurre il tipo di cx

func_for_cx(x);               // chiamata concettuale:
                              // il tipo dedotto per param è il tipo
                              di cx

template<typename T>          // template concettuale per
void func_for_rx(const T&
param);                       // dedurre il tipo di rx
func_for_rx(x);               // chiamata concettuale:
                              // il tipo dedotto per param è il tipo
                              di rx
```

Come abbiamo detto, la deduzione del tipo per `auto` è, con un'unica eccezione che vedremo presto, la stessa cosa della deduzione del tipo per i template.

L'Elemento 1 suddivide la deduzione del tipo per template in tre casi, sulla base delle caratteristiche di *`ParamType`*, lo specificatore di tipo per `param` nel template generale della funzione. In una dichiarazione di variabile che usa `auto`, lo specificatore di tipo prende il posto di *`ParamType`*, e pertanto esistono, anche qui, tre casi.

- Caso 1: lo specificatore di tipo è un puntatore o un riferimento, ma non un riferimento universale.
- Caso 2: lo specificatore di tipo è un riferimento universale.

- Caso 3: lo specificatore di tipo non è né un puntatore né un riferimento.

Abbiamo già visto degli esempi dei Casi 1 e 3:

```
auto x = 27;          // Caso 3 (x non è né puntatore né
                      riferimento)
const auto cx = x;    // Caso 3 (cx non è nessuno dei due)
const auto& rx = x;   // Caso 1 (rx è un riferimento non-universale)
```

Il Caso 2 funziona come ci si può immaginare:

```
auto&& uref1 = x;     // x è int e lvalue,
                      // pertanto il tipo di uref1 è int&

auto&& uref2 = cx;    // cx è const int e lvalue,
                      // pertanto il tipo di uref2 è const int&

auto&& uref3 = 27;    // 27 è int e rvalue,
                      // pertanto il tipo di uref3 è int&&
```

L'Elemento 1 si conclude con una discussione sul fatto che i nomi di array e funzioni decadono in puntatori per gli specificatori di tipo che non sono riferimenti. Ciò accade anche in una deduzione del tipo `auto`:

```
const char name[] =          // il tipo di name è const char[13]
   "R. N. Briggs";

auto arr1 = name;            // il tipo di arr1 è const char*

auto& arr2 = name;           // il tipo di arr2 è
                             // const char (&)[13]

void someFunc(int,
double);                     // someFunc è una funzione;

                             // il tipo è void(int, double)

auto func1 = someFunc;       // il tipo di func1 è
                             // void (*)(int, double)
```

```
auto& func2 = someFunc;       // il tipo di func2 è
                              // void (&)(int, double)
```

Come potete vedere, la deduzione del tipo `auto` funziona come la deduzione del tipo del template. Sono sostanzialmente le due facce della stessa medaglia.

Solo per un aspetto sono differenti. Iniziamo osservando che se si vuole dichiarare un `int` con il valore iniziale `27`, il C++98 fornisce due scelte sintattiche:

```
int x1 = 27;
int x2(27);
```

Il C++11, tramite il suo supporto per l'inizializzazione uniforme, ne aggiunge altre due:

```
int x3 = { 27 };
int x4{ 27 };
```

In sostanza, quattro sintassi per un solo risultato: un `int` con valore `27`.

Ma, come vedremo nell'Elemento 5, vi sono dei vantaggi a dichiarare delle variabili utilizzando `auto` al posto dei tipi fissi, e dunque è comodo sostituire `int` con `auto` nelle dichiarazioni di variabili che abbiamo appena visto. Un'immediata sostituzione testuale fornisce il seguente codice:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

Tutte queste dichiarazioni sono compilabili, ma non hanno lo stesso significato di ciò che sostituiscono. Le prime due istruzioni, in sostanza, dichiarano una variabile di tipo int di valore 27. Ma le altre due dichiarano una variabile di tipo `std::initializer_list<int>` contenente un unico elemento il cui valore è 27!

```
auto x1 = 27;                 // il tipo è int, il valore è 27
auto x2(27);                  // idem
auto x3 = { 27 };             // il tipo è std::initializer_list<int>,
                              // il valore è { 27 }
auto x4{ 27 };                // idem
```

Questo è dovuto a una particolare regola di deduzione del tipo per auto. Quando l'inizializzatore di una variabile dichiarata come `auto` viene racchiuso fra parentesi graffe, il tipo dedotto è `std::initializer_list`. Se tale tipo non può essere dedotto (per esempio perché i valori nell'inizializzatore fra graffe sono di tipo differente), il codice verrà rifiutato:

```
auto x5 = { 1, 2, 3.0 };   // errore! impossibile dedurre T per
                           // std::initializer_list<T>
```

Come indica il commento, la deduzione del tipo non funziona in questo caso, ma è importante riconoscere che, in realtà, si svolgono due diverse deduzioni del tipo. Una deriva dall'uso di `auto`: il tipo di `x5` deve essere dedotto. Poiché l'inizializzatore di `x5` è fra parentesi graffe, `x5` deve essere dedotto come un `std::initializer_list`. Ma `std::initializer_list` è un template. Le istanziazioni sono `std::initializer_list<T>` per un qualche tipo `T` e questo significa che anche il tipo di `T` deve essere dedotto. Tale deduzione rientra nell'ambito del secondo genere di deduzione del tipo che si verifica qui: la deduzione del tipo del template. In questo esempio, tale deduzione non riesce, poiché i valori nell'inizializzatore fra graffe non sono di un unico tipo.

Il trattamento degli inizializzatori fra parentesi graffe è l'unico modo in cui la deduzione del tipo `auto` e la deduzione del tipo del template differiscono. Quando una variabile dichiarata come `auto` viene inizializzata con un inizializzatore fra parentesi graffe, il tipo dedotto è un'istanziazione di `std::initializer_list`. Ma se al template corrispondente viene passato lo stesso inizializzatore, la deduzione del tipo non riesce e il codice viene rifiutato:

```
auto x = { 11, 23, 9 };    // il tipo di x è
                           // std::initializer_list<int>

template<typename T>       // template con parametro
void f(T param);           // dichiarazione equivalente
                           // alla dichiarazione di x

f({ 11, 23, 9 });          // errore! impossibile dedurre il tipo
                           per T
```

Tuttavia, se nel template si specifica che `param` è un `std::initializer_list<T>` per qualche tipo `T` sconosciuto, la deduzione del tipo del template riuscirà a risalire a che cos'è T:

```
template<typename T>
void f(std::initializer_list<T> initList);
```

```
f({ 11, 23, 9 });           // T dedotto come int, e il tipo di initList
                            // è std::initializer_list<int>
```

Pertanto, l'unica vera differenza fra la deduzione del tipo `auto` e template è il fatto che la prima *assume* che un inizializzatore fra parentesi graffe rappresenti un `std::initializer_list`, mentre la deduzione del tipo del template no.

Ci si potrebbe chiedere perché la deduzione del tipo `auto` abbia una regola particolare per gli inizializzatori fra parentesi graffe, mentre la deduzione del tipo del template no. Anch'io me lo chiedo e purtroppo non ho trovato alcuna spiegazione convincente. Ma una regola è una regola e ciò significa che occorre ricordare che se si dichiara una variabile utilizzando `auto` e la si inizializza fra parentesi graffe, il tipo dedotto sarà sempre `std::initializer_list`. Ciò è particolarmente importante da ricordare se si adotta la filosofia dell'inizializzazione uniforme (includendo i valori di inizializzazione fra parentesi graffe). Un classico errore nella programmazione C++11 consiste nel dichiarare accidentalmente una variabile `std::initializer_list` quando invece si voleva dichiarare qualcos'altro. Questa trappola è uno dei motivi per cui alcuni sviluppatori pongono le parentesi graffe attorno agli inizializzatori solo quando sono costretti (parleremo di queste situazioni nell'Elemento 7).

Per il C++11 le cose si fermano qui, mentre per il C++14 il racconto continua. Il C++14 consente ad `auto` di indicare che il tipo restituito da una funzione dovrà essere dedotto (vedi Elemento 3) e le lambda C++14 possono usare `auto` nella dichiarazione dei parametri. Tuttavia, questi utilizzi di `auto` impiegano la *deduzione del tipo del template*, non la deduzione del tipo `auto`. Pertanto una funzione che restituisce un tipo `auto` la quale restituirà un inizializzatore fra parentesi graffe non potrà essere compilata:

```
auto createInitList()
{
```

```
return { 1, 2, 3 };          // errore: impossibile dedurre il tipo
}                            // per { 1, 2, 3 }
```

Lo stesso vale quando `auto` viene utilizzato nella specifica del tipo di un parametro in una lambda C++14:

```
std::vector<int> v;
…

auto resetV =
[&v](const auto& newValue)   // C++14
{ v = newValue; };

…

resetV({ 1, 2, 3 });         // errore! impossibile dedurre il tipo
                             // per { 1, 2, 3 }
```

Argomenti da ricordare

- La deduzione del tipo `auto` è normalmente la stessa cosa della deduzione del tipo del template, ma la prima suppone che un inizializzatore fra parentesi graffe rappresenti un `std::initializer_list`, mentre la seconda no.
- `auto` come tipo restituito da una funzione o parametro lambda implica la deduzione del tipo del template e non la deduzione del tipo `auto`.

Elemento 3 – Come funziona decltype

`decltype` è una creatura un po' particolare. Dato un nome o un'espressione, `decltype` dice qual è il suo tipo. In genere, ciò che dice è esattamente ciò che immaginate. Tuttavia, talvolta fornisce risultati che lasciano perplessi, tanto da rivolgersi, per una "rivelazione", ai siti di riferimento online.

Inizieremo con i casi tipici, quelli che non lasciano spazio a sorprese. Al contrario di ciò che accade durante la deduzione del tipo per i template e `auto`

(Punti 1 e 2), `decltype`, normalmente, replica esattamente il tipo del nome o dell'espressione che avete specificato:

```
const int i = 0;            // decltype(i) è const int

bool f(const Widget& w);    // decltype(w) è const Widget&
                            // decltype(f) è bool(const Widget&)

struct Point {
int x, y;                   // decltype(Point::x) è int
};                          // decltype(Point::y) è int

Widget w;                   // decltype(w) è Widget

if (f(w)) …                 // decltype(f(w)) è bool

template<typename T>        // versione semplificata di std::vector
class vector {
public:
…
T& operator[](std::size_t
index);
…
};

vector<int> v;              // decltype(v) è vector<int>
…
if (v[0] == 0) …            // decltype(v[0]) è int&
```

Avete visto? Nessuna sorpresa.

In C++11, l'utilizzo principale di `decltype` consiste nella dichiarazione di template di funzioni dove il tipo restituito dalla funzione dipende dal tipo dei parametri. Per esempio, supponete di voler scrivere una funzione che prende un container che supporta l'indicizzazione tramite parentesi quadre (ovvero l'uso di "[ ]") più un indice e quindi autentica l'utente prima di restituire il risultato dell'operazione di indicizzazione. Il tipo restituito dalla funzione dovrà essere lo stesso del tipo restituito dall'operazione di indicizzazione.

operator[] su un container di oggetti di tipo `T` restituisce, in genere, un `T&`. Questo è il caso di `std::deque`, per esempio, ed è quasi sempre il caso di `std::vector`. Per `std::vector<bool>`, invece, `operator[]` non restituisce un `bool&`. Al contrario, restituisce un oggetto completamente nuovo. Le motivazioni sono esplorate nell'Elemento 6, ma ciò che è importante è che il tipo restituito dall'**operator[]** di un container dipende dal container stesso.

decltype semplifica il fatto di esprimere tutto questo. Ecco un primo esempio del template che vorremmo scrivere, che mostra l'uso di `decltype` per determinare il tipo restituito. Il template dovrà essere un po' "sistemato", ma lo vedremo più avanti:

```
template<typename Container, typename Index>          // funziona,
auto authAndAccess(Container& c, Index i)             // ma richede
   -> decltype(c[i])                                  // un perfezionamento
{
   authenticateUser();
   return c[i];
}
```

L'uso di `auto` prima del nome della funzione non ha nulla a che vedere con la deduzione del tipo. Piuttosto, indica che viene impiegata la nuova sintassi *trailing return* del C++11: il tipo restituito dalla funzione verrà dichiarato *dopo* l'elenco dei parametri (dopo il "**->**"). Questo ha il vantaggio che per specificare il tipo restituito possono essere utilizzati i parametri della funzione. In `authAndAccess`, per esempio, specifichiamo il tipo restituito utilizzando `c` e `i`. Se il tipo restituito dovesse precedere il nome della funzione come accade nel caso convenzionale, `c` e `i` non sarebbero ancora disponibili, poiché non sarebbero ancora stati dichiarati.

Con questa dichiarazione, `authAndAccess` restituisce il tipo restituito da `operator[]` quando questo viene applicato al container passato: esattamente quello che vogliamo.

Il C++11 fa sì che i tipi restituiti per le lambda mono-istruzione possano essere dedotti e il C++14 estende la possibilità sia a tutte le lambda sia a tutte le funzioni, incluse quelle con più istruzioni. Nel caso di `authAndAccess`, ciò

significa che in C++14 possiamo omettere il tipo finale, lasciando solo l'**auto** iniziale. Con questa forma di dichiarazione, `auto` significa che si verificherà la deduzione del tipo. In particolare, significa che i compilatori dedurranno il tipo restituito dalla funzione dall'implementazione della funzione stessa:

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // non proprio
{ // corretto
   authenticateUser();
   return c[i];                                  // il tipo restituito è dedotto da c[i]
}
```

L'Elemento 2 spiega che per le funzioni in cui viene impiegata la specifica `auto` per il tipo restituito, i compilatori impiegano la deduzione del tipo del template. In questo caso, questo rappresenta un problema. Come abbiamo visto, per la maggior parte dei container-di-**`T`**, **`operator[]`** restituisce un `T&`, ma l'Elemento 1 spiega che durante la deduzione del tipo del template, il fatto che l'espressione di inizializzazione sia un riferimento viene ignorato. Considerate cosa ciò significhi per il seguente codice:

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // autentica l'utente, restituisce d[5],
                           // poi gli assegna 10;
                           // non passa la compilazione!
```

Qui, `d[5]` restituisce un `int&`, la deduzione del tipo restituito `auto` per `authAndAccess` non considererà il fatto che è un riferimento, fornendo pertanto come tipo restituito `int`. Tale `int`, essendo il tipo restituito da una funzione, è un rvalue, e pertanto il codice qui sopra cerca di assegnare 10 a un rvalue `int`. Questo è vietato in C++ e dunque il codice non verrà compilato.

Per far sì che `authAndAccess` funzioni come previsto, occorre utilizzare la deduzione del tipo `decltype` per il suo tipo restituito, ovvero specificare che `authAndAccess` dovrà restituire esattamente lo stesso tipo restituito

dall'espressione `c[i]`. I "custodi" del C++, prevedendo la necessità di utilizzare le regole di deduzione del tipo `decltype` in alcuni casi in cui i tipi devono essere determinati, lo hanno reso possibile in C++14 attraverso lo specificatore `decltype(auto)`. Può sembrare, inizialmente, contraddittorio (`decltype` *e anche* `auto`?), ma tutto ha perfettamente senso: `auto` specifica che il tipo deve essere dedotto e `decltype` dice che per la deduzione devono essere impiegate le regole `decltype`. Pertanto possiamo scrivere `authAndAccess` nel seguente modo:

```
template<typename Container, typename Index>    // C++14; funziona,
decltype(auto)                                   // ma richiede
authAndAccess(Container& c, Index i)             // ancora
{                                                // un perfezionamento
   authenticateUser();
   return c[i];
}
```

Ora, `authAndAccess` restituirà davvero ciò che restituisce `c[i]`. In particolare, per il caso comune in cui `c[i]` restituisca un `T&`, anche `authAndAccess` restituirà un `T&` e nel caso poco comune in cui `c[i]` restituisca un oggetto, anche `authAndAccess` restituirà un oggetto.

L'uso di `decltype(auto)` non è limitato al tipo restituito dalla funzione. Può essere comodo anche per dichiarare le variabili quando si vogliono applicare le regole di deduzione del tipo `decltype` all'espressione di inizializzazione:

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;              // deduzione del tipo auto:
                                  // myWidget1's il tipo è Widget

decltype(auto) myWidget2 = cw;    // deduzione del tipo decltype:
                                  // il tipo di myWidget2 è
                                  // const Widget&
```

Ma qui avete due preoccupazioni. Lo so. Una è il perfezionamento di `authAndAccess` di cui abbiamo parlato, ma che non abbiamo ancora descritto. Ce ne occupiamo ora.

Osservate di nuovo la dichiarazione della versione C++14 di `authAndAccess`:

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

Il container viene passato per riferimento-lvalue-a-non-**`const`,** poiché la restituzione di un riferimento a un elemento del container permette ai client di modificare tale container. Ma questo significa che non è possibile passare dei container rvalue a questa funzione. Gli rvalue non possono legarsi ai riferimenti lvalue (a meno che siano riferimenti lvalue a `const`, cosa che non accade in questo caso).

Effettivamente, il passaggio di un container rvalue ad `authAndAccess` è un caso limite. Un container rvalue, essendo un oggetto temporaneo, verrà normalmente distrutto entro la fine dell'istruzione che contiene la chiamata ad `authAndAccess` e ciò significa che un riferimento a un elemento in tale container (che è tipicamente ciò che restituirà un `authAndAccess`) si perderà alla fine dell'istruzione che lo ha creato. Comunque, ha senso passare ad `authAndAccess` un oggetto temporaneo. Un client potrebbe semplicemente voler eseguire una copia di un elemento nel container temporaneo, per esempio:

```
std::deque<std::string> makeStringDeque();     // funzione factory

// crea una copia del quinto elemento di deque
// restituito da makeStringDeque
auto s = authAndAccess(makeStringDeque(), 5);
```

Per supportare questo utilizzo occorre rivedere la dichiarazione di `authAndAccess` per accettare sia lvalue sia rvalue. L'overloading funzionerebbe (un overload dichiarerebbe un parametro riferimento lvalue e un altro un parametro riferimento rvalue), ma a questo punto occorrerebbe gestire due funzioni. Un modo per evitarlo consiste nel far sì che `authAndAccess` impieghi un parametro riferimento che può legarsi a un lvalue *e* a un rvalue e l'Elemento 24 spiega che questo è esattamente ciò che fanno i riferimenti universali. `authAndAccess` può pertanto essere dichiarato nel seguente modo:

```
template<typename Container, typename Index>    // ora c è
decltype(auto) authAndAccess(Container&& c,      // un riferimento
Index i);                                        // universale
```

In questo template, non sappiamo su quale tipo di container stiamo operando e questo significa che non sappiamo neppure il tipo di oggetti indice che utilizzerà. Impiegando il passaggio per valore di oggetti di un tipo sconosciuto, in genere rischiamo i problemi prestazionali di una copia inutile, il problema comportamentale della riduzione dell'oggetto (vedi Elemento 41) e anche la derisione dei collaboratori, ma, nel caso degli indici del container, seguendo l'esempio della Libreria Standard per i valori indice (in `operator[]` per `std::string`, `std::vector` e `std::deque`), sembra ragionevole: pertanto utilizzeremo comunque il passaggio per valore.

Tuttavia, dobbiamo aggiornare l'implementazione del template sulla base dell'avvertimento presentato nell'Elemento 25 di applicare `std::forward` ai riferimenti universali:

```
template<typename Container, typename Index>    // versione
decltype(auto)                                   // finale
authAndAccess(Container&& c, Index i)            // C++14
{
   authenticateUser();
   return std::forward<Container>(c)[i];
}
```

Questo dovrebbe fare tutto ciò che vogliamo, ma richiede un compilatore C++14. Se non lo avete, dovrete utilizzare la versione C++11 del template. È simile alla versione C++14, tranne per il fatto che dovete specificare da soli il tipo restituito:

```
template<typename Container, typename Index>    // versione
auto                                             // finale
authAndAccess(Container&& c, Index i)            // C++11
-> decltype(std::forward<Container>(c)[i])
{
```

```
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

L'altro problema che probabilmente avete individuato è l'avvertimento posto all'inizio di questo Elemento: che decltype produce *sempre* il tipo previsto, con rare sorprese. In realtà, *difficilmente* incontrerete queste eccezioni alla regola, a meno che vi occupiate di implementazioni di librerie.

Per comprendere *appieno* il comportamento di `decltype`, occorre familiarizzare con alcuni casi speciali. La maggior parte di questi non merita neppure una descrizione in questo libro, ma, osservandone uno, getteremo luce su `decltype` e sul suo uso.

Applicando `decltype` a un nome si ottiene il tipo dichiarato per tale nome. I nomi sono espressioni lvalue, ma questo non riguarda il comportamento di `decltype`. Per le espressioni lvalue più complesse dei nomi, invece, `decltype` garantisce che il tipo restituito sia sempre un riferimento lvalue. Pertanto, se un'espressione lvalue diversa da un nome ha tipo `T`, `decltype` restituisce tale tipo come T&. Raramente questo ha un vero impatto, poiché il tipo della maggior parte delle espressioni lvalue include in modo intrinseco un qualificatore di riferimento lvalue. Le funzioni che restituiscono lvalue, per esempio, restituiscono sempre riferimenti lvalue.

Vi è un'implicazione in questo comportamento che vale la pena di considerare. In

```
int x = 0;
```

`x` è il nome di una variabile, pertanto `decltype(x)` è `int`. Ma includendo il nome `x` fra parentesi (ovvero con "`(x)`"), si ottiene un'espressione più complessa di un nome. Essendo un nome, `x` è un lvalue e il C++ definisce l'espressione `(x)` anch'essa come un `decltype(`**`(x)`**`)` e pertanto `int&`. La collocazione di parentesi attorno a un nome può cambiare il tipo indicato da `decltype`!

In C++11, questo è qualcosa di più di una curiosità, ma insieme al supporto C++14 di `decltype(auto)`, ciò significa che una modifica apparentemente banale nel modo in cui si scrive un'istruzione `return` può influenzare il tipo dedotto per una funzione:

```
decltype(auto) f1()
{
   int x = 0;
   …
   return x;          // decltype(x) è int, pertanto f1 restituisce int
}

decltype(auto) f2()
{
   int x = 0;
   …
   return (x);        // decltype((x)) è int&, pertanto f2 restituisce int&
}
```

Notate che non solo `f2` restituisce un tipo differente da `f1`, ma restituisce anche un riferimento a una variabile locale! Codice di questo tipo ci colloca automaticamente sul binario “alta velocità”, destinazione “comportamento indefinito”, un treno che certamente non volete prendere.

La lezione? Fare particolare attenzione quando si utilizza `decltype(auto)`. Un dettaglio apparentemente insignificante nell’espressione di cui dedurre il tipo può influenzare il tipo restituito da `decltype(auto)`. Per garantire che il tipo dedotto sia quello previsto, utilizzate le tecniche descritte nell’Elemento 4.

Contemporaneamente, non perdete di vista il quadro generale. Certamente, `decltype` (sia da sola sia insieme ad `auto`) può talvolta fornire sorprese in termini di tipo dedotto, ma solo raramente. Normalmente, `decltype` produce il tipo previsto.

Questo in particolare quando `decltype` viene applicato a un nome, poiché in tal caso, `decltype` fa esattamente ciò che ci si può aspettare: restituisce il tipo dichiarato per il nome.

Argomenti da ricordare

- **decltype** restituisce quasi sempre il tipo di una variabile o di un'espressione, senza modificarlo.
- Per le espressioni lvalue di tipo `T` diverso da un nome, `decltype` restituisce sempre il tipo `T&`.
- Il C++14 supporta `decltype(auto)`, che, come `auto`, deduce il tipo dal suo inizializzatore, ma esegue la deduzione del tipo utilizzando le regole di `decltype`.

Elemento 4 – Come visualizzare i tipi dedotti

La scelta degli strumenti per conoscere il risultato nella deduzione del tipo dipende dalla fase del processo di sviluppo software in cui è necessario reperire tale informazione. Esploreremo tre possibilità: ottenere la deduzione del tipo mentre si scrive il codice, durante la compilazione e runtime.

Editor degli IDE

Gli editor di codice, negli IDE, spesso mostrano il tipo delle entità di un programma (variabili, parametri, funzioni e così via) ogni volta che si fa qualcosa come portare il cursore sul tipo delle entità. Per esempio, dato il codice,

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

l'editor di un IDE mostrerà probabilmente che il tipo dedotto per `x` è `int` e per `y` è `const int*`.

Perché questo funzioni, il codice deve essere in uno stato, più o meno, compilabile, poiché ciò che consente all'IDE di offrire questo genere di informazione è un compilatore (o almeno il front-end di un compilatore) che opera all'interno dell'IDE. Se tale compilatore non riesce ad analizzare sufficientemente a fondo il codice, tanto da trarne la deduzione del tipo, non potrà mostrare quale tipo è stato dedotto.

Per i tipi più semplici come `int`, l'informazione fornita dall'IDE è generalmente

corretta. Come presto vedremo, invece, quando sono coinvolti tipi più complessi, l'informazione visualizzata dall'IDE potrebbe non essere particolarmente utile.

Diagnostiche di compilazione

Un modo efficace per far sì che un compilatore mostri il tipo dedotto consiste nell'utilizzare tale tipo in modo che porti a un problema di compilazione. Il messaggio di errore che rileva il problema quasi sicuramente menzionerà il tipo che provoca tale problema.

Supponiamo, per esempio, di voler conoscere il tipo dedotto per x e y nell'esempio precedente. Innanzitutto si dichiara il template di una classe che *non si definisce*. Qualcosa di simile al seguente:

```
template<typename T>       // solo dichiarazione per TD;
class TD;                  // TD == "Type Displayer"
```

I tentativi di istanziare questo template produrranno un messaggio d'errore, poiché non vi è alcuna definizione di template da istanziare. Per vedere i tipi di x e y, basta cercare di istanziare TD con i loro tipi:

```
TD<decltype(x)> xType;     // provoca errori contenenti
TD<decltype(y)> yType;     // i tipi di x e y
```

Utilizzo i nomi di variabili nella forma *variableNameType*, poiché tendono a fornire messaggi d'errore che aiutano a risalire all'informazione ricercata. Per il codice precedente, uno dei compilatori ha fornito le seguenti diagnostiche (dove ho evidenziato le informazioni sul tipo cui eravamo interessati):

```
error: aggregate 'TD<int> xType' has incomplete type and
  cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type and
  cannot be defined
```

Un altro compilatore fornisce questa stessa informazione, in formato un po' differente:

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

A parte le differenze di formattazione, tutti i compilatori provati producono messaggi di errore contenenti informazioni utili, proprio grazie all'impiego di questa tecnica.

Output runtime

L'approccio printf alla visualizzazione delle informazioni sui tipi (non che io vi consigli di utilizzare printf) non può essere impiegato se non runtime, ma offre un pieno controllo sulla formattazione dell'output. Il problema consiste nel creare una rappresentazione testuale del tipo interessato che sia adatta alla visualizzazione. Penserete: "Nessun problema: basta usare `typeid` e `std::type_info::name`". Nel nostro tentativo di trovare i tipi dedotti per `x` e `y`, potreste pensare di scrivere qualcosa come:

```
std::cout << typeid(x).name() << '\n'; // mostra il tipo
std::cout << typeid(y).name() << '\n'; // di x e di y
```

Questo approccio conta sul fatto che chiamando `typeid` su un oggetto come `x` o `y` si ottiene un oggetto `std::type_info`, che ha una funzione membro, `name`, che produce una stringa in stile C (ovvero una `const char*`) che rappresenta il nome del tipo.

Le chiamate a `std::type_info::name` in realtà non necessariamente forniscono qualcosa di sensato, ma le implementazioni tentano di essere utili. Il livello di questo tentativo varia. Per esempio, i compilatori GNU e Clang indicano che il tipo di `x` è "i" e il tipo di `y` è "**`PKi`**". Questi risultati hanno senso una volta che si capisce che, nell'output di questi compilatori, `"i"` significa `"int"` e "**`PK`**" significa " puntatore a `const"` (entrambi i compilatori supportano uno strumento, `c++filt`, che decodifica questi tipi specificati come sigle). Il compilatore Microsoft produce un output più chiaro: `"int"` per `x` e `"int const"`per `y`.

Poiché questi risultati sono corretti per i tipi di `x` e `y`, potreste essere tentati di considerare risolto il problema dell'indicazione del tipo, ma le cose non sono così semplici. Considerate un esempio più complesso:

```
template<typename T>                    // funzione template
void f(const T& param);                 // da richiamare

std::vector<Widget> createVec();        // funzione factory
```

```
const auto vw = createVec();                    // inizializza vw con
                                                 factory

if (!vw.empty()) {
   f(&vw[0]);                                    // richiama f
…
}
```

Questo codice, che impiega un tipo definito dall'utente (`Widget`), un container STL (`std::vector`) e una variabile `auto` (`vw`), è più rappresentativo delle situazioni in cui si vorrebbe avere una certa visibilità sui tipi che vengono dedotti dai compilatori. Per esempio, sarebbe bello sapere quali tipi vengono dedotti per il parametro di tipo del template `T` e per il parametro della funzione `param` in `f`.

Impiegare `typeid` per il problema è semplice. Basta aggiungere del codice a `f` per visualizzare i tipi che vogliamo conoscere:

```
template<typename T>
void f(const T& param)
{
  using std::cout;
  cout << "T = " << typeid(T).name() << '\n'; // mostra T
  cout << "param = " << typeid(param).name() << '\n'; // mostra
  …             // il tipo
}                  // del parametro
```

Gli eseguibili prodotti dai compilatori GNU e Clang producono il seguente output:

```
T = PK6Widget
param = PK6Widget
```

Sappiamo già che per questi compilatori, `PK` significa "puntatore a `const`", dunque l'unico mistero riguarda il numero `6`. Questo è semplicemente il numero di caratteri del nome della classe che segue (`widget`). Pertanto questi compilatori ci dicono entrambi che `T` e `param` sono di tipo `const Widget*`.

Il compilatore Microsoft concorda:

```
T = class Widget const *
param = class Widget const *
```

Tre compilatori indipendenti che producono la stessa informazione suggeriscono che questa informazione sia corretta. Ma osservate più attentamente. Nel template di `f`, il tipo dichiarato per `param` è `const T&`. Considerando questo fatto, non sembra strano che `T` e `param` siano dello stesso tipo? Se `T` fosse `int`, per esempio, il tipo di `param` dovrebbe essere `const int&`, non dello stesso tipo.

Purtroppo i risultati di `std::type_info::name` non sono affidabili. In questo caso, per esempio, il tipo indicato da tutti e tre i compilatori per `param` è errato. Inoltre, sostanzialmente *è necessario* che sia errato, poiché la specifica di `std::type_info::name` indica che il tipo venga trattato come se fosse stato passato a una funzione template come un parametro per valore. Come descritto nell'Elemento 1, ciò significa che se il tipo è un riferimento, questo fatto viene ignorato e se il tipo dopo la rimozione del riferimento è `const` (o `volatile`), il fatto che sia costante (o volatile) debba anch'esso essere ignorato. Questo è il motivo per cui il tipo di `param`, che è `const Widget * const &`, venga indicato come `const Widget*`. Innanzitutto, il fatto che sia un riferimento viene eliminato e poi il fatto che il puntatore risultante sia costante viene anch'esso eliminato.

Pertanto, l'informazione sul tipo indicato dagli editor degli IDE non è da considerarsi affidabile (o almeno non particolarmente utile). Per questo stesso esempio, un editor indica che il tipo di `T` è (non sto scherzando):

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

Lo stesso editor indica che il tipo di `param` è:

```
const std::_Simple_types<...>::value_type *const &
```

Questo spaventa un po' meno del tipo di `T`, ma la presenza dei "... " al centro confonde le cose, prima di capire che è il modo in cui l'editor dell'IDE dice "Sto omettendo tutto ciò che fa parte del tipo di `T`". Con un po' di fortuna, l'ambiente di sviluppo fa un lavoro migliore su codice come questo.

Se preferite contare più sulle librerie che sulla fortuna, sarete felici di sapere che dove `std::type_info::name` e gli IDE non hanno successo, la libreria Boost TypeIndex (spesso indicata come *Boost.TypeIndex)* riesce a ottenere buoni risultati. La libreria non fa parte dello Standard C++, ma neanche l'IDE o template come `TD`. Inoltre, il fatto che le librerie Boost (disponibili in *boost.com)* siano inter-piattaforma, open-source e disponibili sotto una licenza calibrata per risultare accettabile anche al più paranoico dei consulenti legali di un'azienda, significa che il codice che utilizza le librerie Boost è praticamente portatile quanto il codice che conta sulle librerie standard.

Ecco come la funzione `f` può produrre un tipo di informazione accurato utilizzando Boost.TypeIndex:

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // mostra T
  cout << "T = "
    << type_id_with_cvr<T>().pretty_name()
    << '\n';

  // mostra il tipo di param
  cout << "param
  = "
    << type_id_with_cvr<decltype(param)>().pretty_name()
    << '\n';
  …
}
```

Tutto ciò si basa sul fatto che il template di funzione `boost::typeindex::type_id_with_cvr` prende un argomento di tipo (il tipo per il quale vogliamo informazioni) e non elimina i qualificatori `const`, `volatile` o di riferimento (da qui il "`with_cvr`" nel nome del template). Il risultato è un oggetto `boost::typeindex::type_index`, la cui funzione membro `pretty_name` produce una `std::string` contenente una rappresentazione comprensibile del tipo.

Con questa implementazione per `f`, considerate ancora la chiamata che fornisce

informazioni di tipo errato per `param` quando viene impiegato typeid:

```
std::vector<Widget>
createVec();                  // funzione factory

const auto vw =
createVec();                  // inizializza vw con factory

if (!vw.empty()) {
   f(&vw[0]);                 // richiama f
   …
}
```

Qui sotto vediamo i compilatori GNU e Clang; Boost.TypeIndex produce il seguente output, corretto:

```
T = Widget const*
param = Widget const* const&
```

I risultati con il compilatore Microsoft sono sostanzialmente gli stessi:

```
T = class Widget const *
param = class Widget const * const &
```

Questa quasi uniformità è una buona cosa, ma è importante ricordare che gli editor degli IDE, i messaggi d’errore dei compilatori e librerie come Boost.TypeIndex sono semplicemente strumenti che potete utilizzare per cercare di immaginare quali tipi stanno deducendo i compilatori. Possono essere utili, ma alla fine la cosa migliore è capire in quale modo vengono dedotti i tipi, come descritto nei Punti da 1 a 3.

Argomenti da ricordare

- I tipi dedotti possono spesso essere indicati dagli editor degli IDE, dai messaggi d’errore dei compilatori e dalla libreria Boost.TypeIndex.
- I risultati di alcuni strumenti possono non essere né utili né precisi e dunque rimane fondamentale conoscere le regole di deduzione del tipo impiegate dal C++.

2

Il modificatore auto

In teoria, il funzionamento di `auto` è assolutamente semplice, ma il suo comportamento è molto più raffinato di quanto possa sembrare a prima vista. Utilizzandolo si risparmia tempo, certamente, ma si evitano anche i problemi di correttezza e prestazioni che possono affliggere le dichiarazioni manuali del tipo. Inoltre, alcuni dei risultati delle deduzioni del tipo con `auto`, anche se sono fedelmente conformi all'algoritmo prescritto, sono, dal punto di vista del programmatore, semplicemente sbagliate. In questo caso, è importante sapere come guidare `auto` verso la risposta corretta; tornare alle dichiarazioni manuali del tipo è un'alternativa da evitare il più possibile.

Questo breve capitolo descrive tutti i dettagli del modificatore `auto`.

Elemento 5 – Preferire `auto` alle dichiarazioni esplicite del tipo

Ah… quella semplice gioia di un:

```
int x;
```

Aspetta, dannazione, ho dimenticato di inizializzare x, così il suo valore è

indeterminato. Forse. Magari verrà inizializzato a 0. Dipende dal contesto. Oh mio Dio!

Basta preoccuparsi. Ecco la gioia di dichiarare una variabile locale che verrà inizializzata de-referenziando un iteratore:

```
template<typename It>                    // algoritmo per dwim ("fai quello che voglio!")
void dwim(It b, It e)                    // per tutti gli elementi
{                                        // nell'intervallo fra b ed e
   while (b != e) {
   typename std::iterator_traits<It>::value_type
  currValue = *b;
   …
                                         }
   }
}
```

Accidenti! `“typename std::iterator_traits<It>::value_type”` solo per esprimere il tipo di valore puntato da un iteratore? Davvero? A casa mia, la “gioia” è una cosa un po’ diversa. Dannazione. Ma, aspetta, non ne avevamo già parlato?

OK, una semplice gioia (e siamo a tre): la delizia di dichiarare una variabile locale il cui tipo è quello di una *closure*. OK. Il tipo di una closure è conosciuto solo dal compilatore, pertanto non può essere scritto. Ancora una volta, dannazione!

Dannazione, dannazione e dannazione! La programmazione in C++, allora, non è quell’esperienza gioiosa che dovrebbe essere!

In effetti, un tempo era tutto dannatamente complicato. Ma a partire dal C++11 tutti questi problemi spariscono, grazie alla presenza di `auto`. Le variabili `auto` ricevono il proprio tipo tramite deduzione dall’inizializzatore, dunque devono essere inizializzate. Questo significa che potete dire addio a tutti quei problemi legati alla mancata inizializzazione delle variabili, imboccando la grande autostrada del C++ moderno:

```
int x1;                    // potenzialmente non inizializzato

auto x2;                   // errore! Inizializzatore obbligatorio

auto x3 = 0;               // bene, il valore di x è ben definito
```

In ogni autostrada che si rispetti, non si trovano buche, ovvero i problemi legati alla dichiarazione di una variabile locale il cui valore è quello di un iteratore de-referenziato:

```
template<typename It>      // come prima
void dwim(It b, It e)
{
   while (b != e) {
  auto currValue = *b;
  …
   }
}
```

E poiché `auto` usa la deduzione del tipo (vedi Elemento 2), può rappresentare tipi noti solo al compilatore:

```
auto derefUPLess =          // funzione di confronto
   [](const
   std::unique_ptr<Widget>& // per i Widget
   p1,
  const
  std::unique_ptr<Widget>&  // puntati da
  p2)
   { return p1 < p2; };     // std::unique_ptr
```

Molto interessante. In C++14, la situazione migliora ulteriormente, poiché anche i parametri delle espressioni lambda possono impiegare auto:

```
auto derefLess =                        // funzione di confronto
   [](const auto& p1,                   // C++14 per i
```

```
  const auto& p2)                         // valori puntati
   { return p1 < p2; };                   // da qualsiasi cosa
                                          // di tipo puntatore
```

Ciononostante, forse state pensando che non c'è bisogno, davvero, di `auto` per dichiarare una variabile che contiene una closure, poiché possiamo utilizzare un oggetto `std:: function`. Questo è vero, possiamo. Ma forse questo non è ciò che stavate pensando. Forse ciò che state pensando è "… e che cos'è questo oggetto `std::function?"`. Occorre chiarirlo.

`std::function` è un template nella Libreria Standard C++11 che generalizza l'idea di un puntatore a funzione. Mentre i puntatori a funzione possono puntare solo a funzioni, gli oggetti `std::function` possono fare riferimento a ogni oggetto richiamabile, ovvero qualsiasi cosa che possa essere richiamata con una funzione. Così come occorre specificare il tipo della funzione cui puntare quando si crea un puntatore a funzione (per esempio la signature delle funzioni cui si vuole puntare), occorre specificare il tipo di funzione cui fare riferimento quando si crea un oggetto `std::function`. Lo si può fare attraverso il parametro template di `std::function`. Per esempio, per dichiarare un oggetto `std::function` chiamato `func` che potrebbe fare riferimento a qualsiasi oggetto richiamabile e si comporta come se avesse la seguente signature:

```
bool(const
std::unique_ptr<Widget>&,                 // signature C++11 per
  const std::unique_ptr<Widget>&)  // la funzione di confronto
                                   // std::unique_ptr<Widget>
```

si scriverebbe:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)> func;
```

Poiché le espressioni lambda forniscono oggetti richiamabili, le closure possono essere memorizzate in oggetti `std::function`. Ciò significa che si potrebbe dichiarare la versione C++11 di `derefUPLess` senza utilizzare `auto` nel seguente modo:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
```

```
  derefUPLess = [](const std::unique_ptr<Widget>& p1,
                   const std::unique_ptr<Widget>& p2)
                   { return p1 < p2; };
```

È importante riconoscere che anche trascurando la complessità sintattica e la necessità di ripetere i tipi di parametri, l'uso di `std::function` non equivale a usare `auto`. Una variabile dichiarata `auto` contenente una closure ha lo stesso tipo della closure e, in quanto tale, usa esattamente la stessa quantità di memoria richiesta dalla closure. Il tipo di una variabile dichiarata `std::function` che contiene una closure è un'istanziazione del template `std::function` e questo ha dimensioni fisse per una determinata signature. Queste dimensioni possono non essere adeguate per la closure che devono memorizzare e, in questo caso, il costruttore di `std::function` allocherà memoria sullo heap per memorizzare la closure. Il risultato è che l'oggetto `std::function`, in genere, userà più memoria rispetto all'oggetto dichiarato `auto`. E, grazie ai dettagli implementativi che limitano l'inlining e forniscono chiamate indirette a funzione, la chiamata di una closure tramite un oggetto `std::function` è quasi certamente più lenta rispetto a una chiamata attraverso un oggetto dichiarato `auto`. In altre parole, l'approccio con `std::function` è generalmente più "ingombrante" e lento rispetto all'approccio `auto` e può portare a problemi di esaurimento della memoria. In più, come si può vedere negli esempi precedenti, scrivere semplicemente `"auto"` è molto più semplice rispetto a scrivere il tipo dell'istanziazione `std::function`. Nella competizione fra `auto` e `std::function` per tenere una closure, c'è un solo vincitore: `auto`. Un argomento simile può essere fatto per `auto` rispetto a `std::function` per contenere il risultato di chiamate a `std:: bind`, ma nell'Elemento 34, farò del mio meglio per convincervi a utilizzare le lambda invece di `std::bind`.

I vantaggi di `auto` vanno ben oltre la possibilità di evitare le variabili non inizializzate, le dichiarazioni di variabili verbose e la possibilità di trattenere direttamente le closure. Uno è la possibilità di evitare i problemi legati alle "scorciatoie di tipo". Ecco qualcosa che probabilmente avete già visto, e magari anche scritto:

```
std::vector<int> v;
…
unsigned sz = v.size();
```

Il tipo restituito, ufficialmente, da `v.size()` è `std::vector<int>::size_type`,

ma pochi sviluppatori se ne rendono conto. `std::vector<int>::size_type` viene specificato come un tipo intero e senza segno, pertanto molti programmatori immaginano che basti sapere che `std::vector<int>::size_type` è un tipo intero `unsigned` e scrivono del codice come il precedente. Questo può avere alcune interessanti conseguenze. Nell'ambiente Windows a 32 bit, per esempio, `unsigned` e `std::vector<int>::size_type` hanno le stesse dimensioni, mentre in Windows a 64 bit, `unsigned` è di 32 bit, mentre `std::vector<int>::size_type` è di 64 bit. Ciò significa che il codice che funziona in Windows a 32 bit potrebbe comportarsi in modo errato in Windows a 64 bit e, nel passaggio di un'applicazione dai 32 ai 64 bit, chi vuole buttar via del tempo a risolvere problemi di questo tipo?

L'uso di `auto` garantisce di non dover ricorrere a:

```
auto sz = v.size();                    // il tipo di sz è std::vector<int>::size_type
```

Ancora incerti sull'utilità di impiegare `auto`? Allora considerate il codice seguente:

```
std::unordered_map<std::string, int> m;
…
for (const std::pair<std::string, int>& p : m) {
{
  …         // fa qualcosa con p
}
```

Sembra perfettamente accettabile, ma contiene un problema. Riuscite a individuarlo?

Occorrerà ricordarsi che l'elemento chiave di `std::unordered_map` è `const`, pertanto il tipo di `std::pair` nella tabella hash (quale è `std::unordered_map`), non è `std:: pair<std::string, int>`, è `std::pair<const std::string, int>`. Ma questo non è il tipo dichiarato per la variabile `p` nel ciclo. Come risultato, i compilatori cercheranno di trovare un modo per convertire gli oggetti `std::pair<const std::string, int>` (ovvero il contenuto della tabella hash) in oggetti `std::pair<std::string, int>` (il tipo dichiarato per `p`). Vi riusciranno creando un oggetto temporaneo del tipo cui `p` vuole legarsi copiando ciascun oggetto in `m`, poi collegando il riferimento `p` a questo oggetto temporaneo. Alla fine di ciascuna iterazione del ciclo, l'oggetto temporaneo

verrà distrutto. Se provate a utilizzare questo ciclo, probabilmente resterete sorpresi dal suo comportamento, poiché, quasi certamente, intendevate semplicemente legare il riferimento `p` a ciascun elemento di `m`.

Questa differenza indesiderata di tipo può essere eliminata con `auto`:

```
for (const auto& p : m)
{
  …           // come prima
}
```

Non solo questo è più efficiente, ma è anche più facile da scrivere. Inoltre, questo codice presenta il grande vantaggio che se si prende l'indirizzo di `p`, si è sicuri di ottenere un puntatore a un elemento all'interno di `m`. Nel codice che non usa `auto`, si otterrebbe un puntatore a un oggetto temporaneo, un oggetto che verrà poi distrutto, alla fine dell'iterazione del ciclo.

Gli ultimi due esempi, scrivere `unsigned` al posto di `std::vector<int>::size_type` e scrivere `std::pair<std::string, int>` al posto di `std::pair<const std::string, int>`) dimostrano come il fatto di specificare esplicitamente il tipo possa portare a conversioni implicite indesiderate e imprevedibili. Se si utilizza `auto` come tipo della variabile in questione, non è necessario preoccuparsi delle differenze fra il tipo della variabile dichiarata e il tipo dell'espressione utilizzata per inizializzarla.

Vi sono pertanto vari motivi che spingono a preferire `auto` rispetto a una dichiarazione esplicita del tipo. Tuttavia, neppure `auto` è perfetto. Il tipo di ciascuna variabile `auto` viene dedotto dalla sua espressione di inizializzazione e alcune espressioni di inizializzazione sono di un tipo imprevedibile e indesiderabile. Le condizioni in cui sorgono tali casi e che cosa si può fare al riguardo sono argomenti trattati negli Elementi 2 e 6, quindi non è il caso di ripetersi qui. Rivolgeremo invece l'attenzione a un altro problema che può sorgere utilizzando `auto` al posto di una dichiarazione tradizionale del tipo: la leggibilità del codice sorgente risultante.

Innanzitutto, fate un bel respiro e tranquillizzatevi. Usare `auto` è una scelta, non un obbligo. Se, secondo la vostra opinione professionale, il vostro codice sarebbe più chiaro o più facile da rielaborare o, per qualche altro motivo, migliore utilizzando dichiarazioni esplicite del tipo, sentitevi liberi di continuare a farlo. Ma tenete in considerazione che il C++ non introduce nulla di nuovo in ciò che, nel mondo dei linguaggi di programmazione, viene generalmente

chiamato con il termine di *inferenza del tipo*. Altri linguaggi procedurali con tipo statico (per esempio C#, D, Scala, Visual Basic) hanno una funzionalità più o meno equivalente, per non parlare di un'intera varietà di linguaggi funzionali con tipo statico (per esempio ML, Haskell, OCaml, F# e così via). In parte, questo è dovuto al successo dei linguaggi a tipo dinamico, come Perl, Python e Ruby, dove le variabili ricevono raramente un tipo esplicito. La comunità degli sviluppatori software ha una grande esperienza nel campo dell'inferenza del tipo e ha dimostrato che non vi è nulla di contradditorio fra questa tecnologia e la creazione e manutenzione di grandi basi di codice di livello professionale.

Alcuni sviluppatori sono disturbati dal fatto che l'utilizzo di `auto` impedisce di determinare il tipo di un oggetto semplicemente osservando il codice sorgente. Tuttavia, la capacità dell'IDE di mostrare il tipo di un oggetto, spesso allevia questo problema (anche tenendo in considerazione i problemi di visualizzazione del tipo negli IDE, menzionati nell'Elemento 4) e, in molti casi, un'indicazione astratta del tipo dell'oggetto può essere utile quanto un'indicazione esatta. Spesso basta, per esempio, sapere che un oggetto è un container o un contatore o un puntatore smart, senza sapere esattamente quale tipo di container, contatore o puntatore smart sia. Adottando nomi di variabile ben scelti, questa indicazione astratta del tipo può essere altrettanto "parlante".

Il fatto è che, scrivendo esplicitamente il tipo, spesso si fa qualcosa di più che introdurre possibilità che sorgano errori subdoli, in termini sia di correttezza sia di efficienza (o di entrambe le cose). Inoltre, i tipi di `auto` cambiano automaticamente quando cambia il tipo dell'espressione utilizzata per la loro inizializzazione e ciò significa che alcune rielaborazioni vengono facilitate dall'uso di `auto`. Per esempio, se una funzione è dichiarata in modo che restituisca un `int`, ma successivamente si decide che sarebbe meglio impiegare un `long`, il codice chiamante si aggiornerà da solo alla successiva compilazione se i risultati della chiamata della funzione vengono memorizzati in variabili `auto`. Se invece i risultati vengono memorizzati in variabili dichiarate esplicitamente come `int`, sarete costretti a trovare tutte le chiamate per correggerle.

Argomenti da ricordare

- Le variabili `auto` devono essere inizializzate, sono generalmente immuni

agli errori di tipo che portano a problemi di portabilità o efficienza, possono facilitare il processo di rielaborazione del codice e, in genere, richiedono meno lavoro alla tastiera rispetto alle variabili il cui tipo è specificato esplicitamente.

- Le variabili con tipo `auto` sono però soggette alle trappole descritte negli Elementi 2 e 6.

Elemento 6 – Uso di un inizializzatore di tipo esplicito quando `auto` deduce tipi indesiderati

Come spiega l'Elemento 5, impiegando `auto` per dichiarare le variabili si ottengono vari vantaggi tecnici rispetto all'indicazione esplicita del tipo, ma talvolta la deduzione del tipo provocata da `auto` è imprecisa. Per esempio, supponete di avere una funzione che prende un `Widget` e restituisce un `std::vector<bool>`, dove ciascun `bool` indica se `Widget` offre una determinata funzionalità:

```
std::vector<bool> features(const Widget& w);
```

Inoltre, supponete che il bit 5 indichi se il `Widget` ha una priorità elevata. Possiamo pertanto scrivere il seguente codice:

```
Widget w;
…
bool highPriority = features(w)[5];      // w è ad alta priorità?
…
processWidget(w, highPriority);           // elabora w in base
                                          // alla sua priorità
```

Non c'è niente di sbagliato in questo codice. Funzionerà perfettamente. Ma se eseguiamo una modifica apparentemente innocua, sostituendo `auto` al tipo esplicito di `highPriority`,

```
auto highPriority = features(w)[5];      // w è ad alta priorità?
```

la situazione cambia. Il codice continuerà a risultare compilabile, ma il suo comportamento non sarà più prevedibile:

```
processWidget(w, highPriority); // comportamento indefinito!
```

Come è indicato dal commento, la chiamata a `processWidget` ora ha un comportamento indefinito. Perché? La risposta sarà un po' sorprendente. Nel codice che utilizza `auto`, il tipo di `highPriority` non è più `bool`. Anche se `std::vector<bool>` contiene concettualmente `bool`, `operator[]` per `std::vector<bool>` non restituisce un riferimento a un elemento del container (che è ciò che `std::vector::operator[]` restituisce *per ogni tipo ad eccezione di* `bool`). Al contrario, restituisce un oggetto di tipo `std::vector<bool>::reference` (una classe nidificata all'interno di `std::vector<bool>`).

`std::vector<bool>::reference` esiste poiché `std::vector<bool>` è specificato per rappresentare i suoi `bool` in una forma "a pacchetto", un bit per `bool`. Ciò crea un problema per l'**`operator[]`** di `std::vector<bool>`, in quanto `operator[]`, per `std::` vector<T>, si suppone che restituisca un `T&`, mentre il C++ proibisce i riferimenti ai bit. Non potendo restituire un `bool&`, `operator[]` per `std::vector<bool>` restituisce un oggetto che si *comporta come* un `bool&`. Perché questo meccanismo possa funzionare, gli oggetti `std::vector<bool>::reference` devono essere utilizzabili sostanzialmente in tutti i contesti in cui possono essere utilizzati `bool&`. Fra le funzionalità di `std::vector<bool>::reference` che rendono possibile tutto questo, vi è una conversione implicita in `bool`. (Non in `bool&`, ma in *`bool`*. Spiegare tutte le tecniche impiegate da `std::vector<bool>::reference` per emulare il comportamento di un `bool&` richiederebbe troppo spazio, quindi diciamo semplicemente che questa conversione implicita è solo un'importante tessera di un mosaico ben più grande.)

Tenendo in considerazione questa informazione, osserviamo ancora una volta questa parte del codice originale:

```
bool highPriority = features(w)[5];      // dichiara esplicitamente
                                         // il tipo di highPriority
```

Qui, `features` restituisce un oggetto `std::vector<bool>`, sul quale viene richiamato `operator[]`. `operator[]` restituisce un oggetto

`std::vector<bool>::reference`, che viene poi convertito implicitamente in `bool`, che è necessario per inizializzare `highPriority`. `highPriority`, pertanto, finisce per contenere il valore del bit 5 di `std::vector<bool>` restituito da `features`, esattamente come previsto.

Ecco invece che cosa accade nella dichiarazione con `auto` di highPriority:

```
auto highPriority = features(w)[5];      // deduce il tipo
                                         // di highPriority
```

Ancora una volta, `features` restituisce un oggetto `std::vector<bool>` e, ancora una volta, `operator[]` viene richiamato su di esso. `operator[]` continua a restituire un oggetto `std::vector<bool>::reference`, ma ora vi è una differenza, poiché `auto` lo deduce come tipo di `highPriority`. `highPriority` non contiene il valore del bit 5 del `std::vector<bool>` restituito da `features`.

Il valore che contiene dipende dal modo in cui è implementato `std::vector<bool>::reference`. Un'implementazione prevede che tali oggetti contengano un puntatore alla word che contiene il bit interessato, più l'offset, all'interno di tale word, per raggiungere questo bit. Considerate ciò che significa per l'inizializzazione di `highPriority`, supponendo che sia attiva tale implementazione di `std::vector<bool>::reference`.

La chiamata a `features` restituisce un oggetto `std::vector<bool>` temporaneo. Questo oggetto non ha alcun nome, ma per gli scopi di questa discussione, lo chiameremo *`temp`*. operator[] viene richiamato su *`temp`* e il `std::vector<bool>::reference` che restituisce contiene un puntatore a una word nella struttura dei dati che contiene i bit gestiti da *`temp`*, più l'offset all'interno di tale word corrispondente al bit 5. `highPriority` è una copia di questo oggetto `std::vector<bool>::reference`, così anche `highPriority` contiene un puntatore a una word in *`temp`*, più l'offset corrispondente al bit 5. Alla fine dell'istruzione, *`temp`* viene distrutto, poiché si tratta di un oggetto temporaneo. Pertanto, `highPriority` contiene un puntatore pendente e questo è il motivo del comportamento indefinito nella chiamata a processWidget:

```
processWidget(w, highPriority);   // comportamento indefinito!
                                  // highPriority contiene
                                  // un puntatore pendente!
```

std::vector<bool>::reference è un esempio di *classe proxy:* una classe che esiste con lo scopo di emulare ed estendere il comportamento di qualche altro tipo. Le classi proxy vengono impiegate per vari scopi. std::vector<bool>::reference esiste per offrire l'illusione che operator[] per std::vector<bool> restituisca un riferimento a un bit, per esempio, e i tipi per puntatori smart della Libreria Standard (vedere il Capitolo 4) sono classi proxy che applicano la gestione delle risorse ai comuni puntatori. L'utilità delle classi proxy è indiscutibile. In pratica, lo stesso concetto di "proxy" è molto affermato nel "Pantheon" dello sviluppo software.

Alcune classi proxy sono progettate per essere visibili ai client. Questo è il caso di std:: shared_ptr e std::unique_ptr, per esempio. Altre classi proxy sono progettate per comportarsi in modo più o meno invisibile. std::vector<bool>::reference è un esempio di questi proxy "invisibili", così come il suo compagno std::bitset::reference.

Inoltre, sul terreno vi sono alcune classi delle librerie C++ che impiegano una tecnica nota come *expression template*. Tali librerie sono state originariamente sviluppate per migliorare l'efficienza del codice numerico. Per esempio, data una classe Matrix e degli oggetti m1, m2, m3 e m4 di tipo Matrix, l'espressione

```
Matrix sum = m1 + m2 + m3 + m4;
```

può essere calcolata in modo molto più efficiente se operator+ per gli oggetti Matrix restituisce un proxy per il risultato invece del risultato stesso. In pratica, operator+ per due oggetti Matrix restituirà un oggetto di una classe Sum<Matrix, Matrix> invece che un oggetto di tipo Matrix. Come è stato il caso di std::vector<bool>::reference e bool, vi sarebbe una conversione implicita dalla classe proxy a Matrix, che permetterebbe l'inizializzazione di sum dall'oggetto proxy prodotto dall'espressione che si trova sul lato destro del segno "=". Il tipo di tale oggetto codificherebbe tradizionalmente l'intera espressione di inizializzazione, ovvero sarebbe qualcosa come Sum<Sum<Sum<Matrix, Matrix> che è certamente un tipo da cui i client dovrebbero essere protetti.

Come regola generale, le classi proxy "invisibili" non vanno troppo d'accordo con auto. Gli oggetti di queste classi, in genere, non sono progettati per vivere più a lungo di un'unica istruzione e dunque la creazione di variabili di questi tipi

tende a violare le assunzioni progettuali fondamentali della libreria. Questo è il caso di `std::vector<-bool>::reference` e abbiamo visto che violare tale assunto può portare a comportamenti indefiniti.

Pertanto si deve evitare di scrivere codice nella seguente forma:

```
auto someVar = espressione di un tipo classe proxy "invisibile";
```

Ma come si può riconoscere quando sono attivi degli oggetti proxy? Il software che li impiega non mostra affatto la loro esistenza. Si suppone che siano invisibili, almeno concettualmente! E una volta che li abbiamo trovati, dobbiamo davvero abbandonare `auto` e i tanti vantaggi evidenziati dall'Elemento 5 solo per questo motivo?

Affrontiamo innanzitutto il problema di trovarli. Anche se le classi proxy "invisibili" sono progettate per non comparire nel "radar" del programmatore nell'utilizzo quotidiano, le librerie che le utilizzano spesso ne documentano la presenza. Più familiarizzerete con le decisioni progettuali che stanno alla base delle librerie che impiegate, più saprete quale uso viene fatto dei proxy in queste librerie.

Quando anche la documentazione non aiuta, basta cercare nei file header. Raramente il codice sorgente riesce a nascondere completamente l'impiego di oggetti proxy. Vengono normalmente restituiti da funzioni che si prevede che vengano chiamate dai client, dunque la signature della funzione normalmente riflette la loro esistenza. Ecco, per esempio, l'aspetto di `std::vector<bool>::operator[]`:

```
namespace std {         // standard C++

  template <class Allocator>
  class vector<bool, Allocator> {
  public:
    …
    class reference { … };

    reference operator[](size_type n);
    …
  };
}
```

Supponendo di sapere che `operator[]` per `std::vector<T>` normalmente

restituisce un `T&`, il tipo non convenzionale restituito per `operator[]` in questo caso è un indizio dell'impiego di una classe proxy. Facendo molta attenzione alle interfacce utilizzate, spesso è possibile individuare l'esistenza delle classi proxy.

In pratica, molti sviluppatori scoprono l'uso delle classi proxy solo quando tentano di venire a capo di strani problemi di compilazione o di eseguire il debugging di risultati errati dei test. Indipendentemente dal modo in cui le trovate, una volta che avete determinato che `auto` deduce il tipo di una classe proxy invece del tipo cui fa riferimento la classe proxy, la soluzione non prevede necessariamente l'abbandono di `auto`. Il problema non è, infatti, dovuto all'**`auto`.** Il problema è che `auto` non deduce il tipo che noi vogliamo che deduca. La soluzione consiste nel costringerlo a impiegare un'altra deduzione del tipo. Il modo per farlo può essere chiamato *inizializzatore con tipo esplicito.*

Un inizializzatore con tipo esplicito prevede la dichiarazione di una variabile con `auto`, convertendo però l'espressione di inizializzazione nel tipo che `auto` dovrà poi dedurre.

Ecco come si può utilizzare questa sintassi per costringere `highPriority` a essere, per esempio, un `bool`:

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

Qui, `features(w)[5]` continua a restituire a un oggetto `std::vector<bool>::reference`, come ha sempre fatto, ma la conversione (`static_cast`) cambia il tipo dell'espressione in un `bool`, che `auto`, poi, deduce essere il tipo di `highPriority`. Runtime, l'oggetto `std::vector<bool>::reference` restituito da `std::vector<-bool>::operator[]` esegue la conversione in `bool` che supporta e, nell'ambito di tale conversione, viene referenziato il, comunque valido, puntatore a `std::vector<bool>` restituito da `features`. Ciò evita il comportamento indefinito che abbiamo visto in precedenza. Ai bit puntati dal puntatore viene poi applicato l'indice 5 e il valore `bool` che ne emerge viene infine utilizzato per inizializzare `highPriority`.

Quanto all'esempio di `Matrix`, l'inizializzatore con tipo esplicito avrebbe il seguente aspetto:

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

Gli usi di questa forma non si limitano agli inizializzatori che forniscono tipi costituiti da classi proxy. Possono essere utili anche per chiarire il fatto che si sta creando deliberatamente una variabile di un tipo che è differente da quello generato dall'espressione di inizializzazione. Per esempio, supponete di avere una funzione per calcolare un determinato valore di tolleranza:

```
double
calcEpsilon();          // restituisce il valore della tolleranza
```

`calcEpsilon` restituisce chiaramente un `double`, ma supponete di sapere che per la vostra applicazione sia adeguata la precisione di un `float` e che per voi conti la differenza dimensionale esistente fra un `float` e un `double`. Potreste dichiarare una variabile `float` per memorizzare il risultato di `calcEpsilon`,

```
float ep = calcEpsilon();  // conversione implicita
                           // double → float
```

Ma questo non comunica esattamente "Sto riducendo deliberatamente la precisione del valore restituito dalla funzione". Una dichiarazione che utilizza l'inizializzatore con tipo esplicito, invece, chiarisce questo fatto:

```
auto ep = static_cast<float>(calcEpsilon());
```

Si applica un ragionamento simile quando si ha un'espressione in virgola mobile che si memorizza deliberatamente sotto forma di un valore intero. Supponete di dover calcolare gli indici di un elemento in un container con degli iteratori ad accesso casuale (per esempio un `std::vector`, `std::deque` o `std::array`) e di ricevere un `double` compreso fra `0.0` e `1.0` che indica la distanza rispetto all'inizio del container in cui è situato l'elemento desiderato (`0.5` indicherebbe, quindi, il centro del container). Inoltre, supponete di pensare che l'indice risultante sia adatto alla memorizzazione in un `int`. Se il container è `c` e il double è `d`, potreste calcolare l'indice nel seguente modo,

```
int index = d * c.size();
```

ma ciò nasconderebbe il fatto che state intenzionalmente convertendo il `double` a destra in modo che diventi un int. L'inizializzatore con tipo esplicito chiarisce questo fatto:

```
auto index = static_cast<int>(d * c.size());
```

Argomenti da ricordare

- I tipi proxy “invisibili” possono far sì che `auto` deduca il tipo “errato” per un’espressione di inizializzazione.
- L’inizializzatore con tipo esplicito costringe `auto` a dedurre il tipo desiderato.

3

Passare al C++ moderno

In termini di funzionalità "importanti" il C++11 e il C++14 hanno molto da offrire. auto, i puntatori smart, le semantiche di spostamento, le lambda, la concorrenza: tutti argomenti di grande importanza. Per questo meritano un intero capitolo. È fondamentale conoscere esattamente il funzionamento di queste novità, ma, per diventare davvero efficaci nell'arte della programmazione C++ moderna, è necessario anche procedere a piccoli passi. Ogni passo risponde a specifiche domande che sorgono nel passaggio dal C++98 al C++ moderno. Quando si dovrebbero utilizzare le parentesi graffe al posto delle parentesi standard per la creazione degli oggetti? Perché le dichiarazioni alias sono migliori rispetto ai `typedef`? Che differenza c'è tra `constexpr` e `const`? Qual è la relazione esistente fra le funzioni membro `const` e la sicurezza del thread? E l'elenco è lungo. Un passo dopo l'altro, questo capitolo fornisce tutte le risposte.

Elemento 7 – Distinguere fra () e {} nella creazione degli oggetti

In base al vostro punto di vista, le scelte sintattiche per l'inizializzazione degli oggetti in C++11 possono lasciare l'imbarazzo della scelta o una grande confusione. Come regola generale, i valori di inizializzazione possono essere

specificati fra parentesi, col segno “=” o fra parentesi graffe:

```
int x(0);                          // inizializzatore fra parentesi

int y = 0;                         // inizializzatore dopo "="

int z{ 0 };                        // inizializzatore fra graffe
```

In molti casi è anche possibile utilizzare, insieme, il segno di uguaglianza e le parentesi graffe:

```
int z = { 0 };                     // inizializzatore dopo "=" e fra
                                   graffe
```

In questo Elemento, in generale, ignorerò quest’ultima sintassi (segno uguale più parentesi graffe), poiché il C++, normalmente, la tratta allo stesso modo della versione con le sole parentesi graffe.

La “grande confusione” deriva dal fatto che, spesso, l’uso del segno di uguaglianza per l’inizializzazione inganna chi non è esperto di programmazione C++, facendo pensare che si stia svolgendo un assegnamento, mentre non è così. Per i tipi interni, come `int`, la differenza è puramente accademica, mentre per i tipi definiti dall’utente, è importante distinguere fra inizializzazione e assegnamento, poiché sono previste chiamate a funzione differenti:

```
Widget w1;                         // richiama il costruttore di
                                   default

Widget w2 = w1;                    // non assegnamento; richiama
                                   costr. per copia

w1 = w2;                           // assegnamento; richiama
                                   l'operatore = per copia
```

Anche avendo a disposizione più sintassi di inizializzazione, vi sono situazioni in cui il C++98 non aveva alcun modo per esprimere l’inizializzazione desiderata. Per esempio, non consentiva di indicare direttamente la necessità di creare un container STL che contenga uno specifico set di valori (per esempio `1`, `3` e `5`).

Per risolvere la confusione delle sintassi multiple di inizializzazione e anche il fatto che esse non coprivano, comunque, tutte le situazioni di inizializzazione, il C++11 introduce l'*inizializzazione uniforme*: un'unica sintassi di inizializzazione che può, almeno teoricamente, essere utilizzata ovunque per esprimere qualsiasi cosa. Si basa sulle parentesi graffe e per questo motivo molti preferiscono chiamarla *inizializzazione a graffe*. Tuttavia "inizializzazione uniforme" rappresenta un concetto, mentre "inizializzazione a graffe" rappresenta un costrutto sintattico.

L'inizializzazione a graffe consente di esprimere ciò che in precedenza non poteva essere espresso. Utilizzando le parentesi graffe, è facile specificare il contenuto iniziale di un container:

```
std::vector<int> v{ 1, 3, 5};     // il contenuto iniziale di v è
                                  1, 3, 5
```

Le parentesi graffe possono essere utilizzate anche per specificare i valori di inizializzazione di default per i dati membro non statici. Questa possibilità, introdotta dal C++11, è condivisa con la sintassi di inizializzazione "=" ma non con le parentesi:

```
class Widget {
  …
private:
  int x{ 0 };                     // funziona, il valore di default
                                  di x è 0
  int y = 0;                      // funziona
  int z(0);                       // errore!
};
```

Al contrario, gli oggetti non copiabili (per esempio `std::atomic`, vedi Elemento 40) possono essere inizializzati utilizzando le parentesi graffe o le parentesi standard, ma non utilizzando "=":

```
std::atomic<int> ai1{ 0 };        // OK

std::atomic<int> ai2(0);          // OK
```

```
std::atomic<int> ai3 = 0;          // Errore!
```

Pertanto è facile capire perché l’inizializzazione a graffe viene chiamata “uniforme”. Dei tre modi impiegati dal C++ per designare un’espressione di inizializzazione, solo quella a graffe può essere impiegata ovunque.

Una nuova funzionalità dell’inizializzazione a graffe è il fatto che impedisce le *conversioni narrowing* implicite fra i tipi standard. Se non si può garantire che il valore di un’espressione in un inizializzatore a graffe possa essere esprimibile dal tipo dell’oggetto inizializzato, il codice non verrà compilato:

```
double x, y, z;
…
int sum1{ x + y + z };             // Errore! Una somma di double potrebbe
                                   // non essere esprimibile come int
```

L’inizializzazione a parentesi e con “=” non controlla le conversioni narrowing, poiché ciò potrebbe rendere inutilizzabile troppo codice preesistente:

```
int sum2( x + y + z);              // OK (il valore dell'espressione
                                   // è troncato in un int)

int sum3 = x + y + z;              // Idem
```

Un’altra caratteristica degna di nota dell’inizializzazione a graffe è la sua immunità al problema del *most vexing parse* del C++. Si tratta di un effetto collaterale della regola del C++ in base alla quale tutto ciò che può essere considerato come una dichiarazione debba essere interpretato insieme: il problema del *most vexing parse* affligge gli sviluppatori soprattutto quando vogliono applicare il costruttore di default di un oggetto, ma finiscono inavvertitamente per dichiarare una funzione. La radice del problema è il fatto che, se si vuole richiamare un costruttore con un argomento, lo si può fare nel seguente modo,

```
Widget w1(10);                     // richiama il costr. di Widget con argomento 10
```

ma se si cerca di richiamare un costruttore di `Widget` con zero argomenti utilizzando una sintassi analoga, si dichiara una funzione, invece di un oggetto:

```
Widget w2();                      // most vexing parse! dichiara
                                  una funzione
                                  // di nome w2 che restituisce un
                                  Widget!
```

Le funzioni non possono essere dichiarate utilizzando le parentesi graffe per l'elenco dei parametri, pertanto, costruendo per default un oggetto impiegano le parentesi graffe non si ha questo problema:

```
Widget w3{};                      // richiama il costr. di Widget
                                  senza argomenti
```

Vi è molto da dire sull'inizializzazione a graffe. È la sintassi che può essere utilizzata nella più grande varietà di contesti, previene le conversioni implicite narrowing ed è immune al problema del most vexing parse del C++. Tre grandi vantaggi! Allora, perché questo Elemento non si chiama semplicemente "Applicate a tappeto la sintassi di inizializzazione a graffe"?

Il difetto dell'inizializzazione a graffe è il talvolta sorprendente comportamento che l'accompagna. Tale comportamento deriva dalla relazione particolarmente difficile fra gli inizializzatori a graffe, `std::initializer_list` e la risoluzione dell'overload del costruttore. Le loro interazioni possono portare a codice che sembra fare una certa cosa, mentre in realtà ne fa un'altra. Per esempio, l'Elemento 2 spiega che quando una variabile dichiarata `auto` ha un inizializzatore a graffe, il tipo dedotto è `std::initializer_list`, anche se altri modi per dichiarare una variabile con lo stesso inizializzatore fornirebbero un tipo più intuitivo. Il risultato è che più si apprezza `auto`, meno si è entusiasti dell'inizializzazione a graffe.

Nelle chiamate al costruttore, le parentesi e le parentesi graffe hanno lo stesso significato, sempre che non siano coinvolti i parametri `std::initializer_list`:

```
class Widget {
public:
  Widget(int i, bool b);          // costr. che non dichiarano
  Widget(int i, double d);        // parametri
                                  std::initializer_list
```

```
  …
};

Widget w1(10, true);            // richiama il primo costruttore

Widget w2{10, true};            // richiama il primo costruttore

Widget w3(10, 5.0);             // richiama il secondo
                                costruttore

Widget w4{10, 5.0};             // richiama il secondo
                                costruttore
```

Se, invece, uno o più costruttori dichiarano un parametro di tipo `std::initializer_list`, le chiamate che usano la sintassi di inizializzazione a graffe preferiscono decisamente gli overload con `std::initializer_list`. *Decisamente*. Se esiste un *qualsiasi modo* per cui i compilatori costruiscano una chiamata utilizzando un inizializzatore a graffe che sia un costruttore che prende un `std::initializer_list`, i compilatori seguiranno tale interpretazione. Se la classe `Widget` qui sopra viene estesa con un costruttore che accetta, per esempio, un `std::initializer_list<long double>`,

```
class Widget {
public:
  Widget(int i, bool b);             // come prima
  Widget(int i, double d);           // come prima
  Widget(std::initializer_list<long  // aggiunta
  double> il);

  …
};
```

i Widget w2 e w4 verranno costruiti utilizzando il nuovo costruttore, anche se il tipo degli elementi `std::initializer_list` (`long double`) è, rispetto ai costruttori non `std::initializer_list`, una scelta di secondo piano per entrambi gli argomenti! Osservate:

```
Widget w1(10, true);          // usa le parentesi e, come prima,
                              // richiama il primo costruttore

Widget w2{10, true};          // usa le graffe, ma ora richiama
                              // std::initializer_list ctor
                              // (10 e true convertiti in long double)

Widget w3(10, 5.0);           // usa le parentesi e, come prima,
                              // richiama il secondo costruttore

Widget w4{10, 5.0};           // usa le graffe, ma ora richiama
                              // std::initializer_list ctor
                              // (10 e 5.0 convertiti in long double)
```

Anche ciò che sarebbe normalmente un costruttore per copia e per spostamento, può essere trasformato in un costruttore `std::initializer_list`:

```
class Widget {
public:
  Widget(int i, bool b);                            // come prima
  Widget(int i, double d);                          // come prima
  Widget(std::initializer_list<long double> il);    // come prima

operator float() const;                             // converte

  …                                                 // in float
};

Widget w5(w4);                  // usa le parentesi, richiama il costr. per copia
```

```
Widget w6{w4};                    // usa le graffe, richiama
                                  // std::initializer_list ctor
                                  // (w4 convertito in float, e
                                  float
                                  // convertito in long double)

Widget w7(std::move(w4));          // usa le parentesi, richiama
                                  il costr. per spostamento
Widget w8{std::move(w4)};          // usa le graffe, richiama
                                  // std::initializer_list ctor
                                  // (per lo stesso motivo di w6)
```

La determinazione dei compilatori di legare gli inizializzatori a graffe ai costruttori che accettano `std::initializer_list` è così forte che prevale anche se il costruttore `std::initializer_list` più adatto non può essere richiamato. Per esempio:

```
class Widget {
public:
  Widget(int i, bool b);             // come prima
  Widget(int i, double d);           // come prima

  Widget(std::initializer_list<bool>
  il);                               // ora il tipo
                                     // dell'elemento è bool

  …                                  // nessuna funzione
};                                   // di conversione implicita


Widget w{10, 5.0};                   // errore! richiede
                                     conversioni narrowing
```

Qui, i compilatori ignoreranno i primi due costruttori (il secondo dei quali offre una corrispondenza esatta per entrambi i tipi di argomenti) e tenterà di chiamare

il costruttore che prende un `std::initializer_list<bool>`. La chiamata di tale costruttore richiederebbe la conversione di un int(10) e di un double(5.0) in bool. Entrambe le conversioni sarebbero in riduzione (`bool` non può certo rappresentare esattamente questi valori) e le conversioni narrowing sono proibite all'interno degli inizializzatori a graffe; dunque la chiamata non è valida e il codice viene rifiutato.

Solo se non vi è alcun modo per convertire i tipi di argomenti in un inizializzatore a graffe al tipo in `std::initializer_list`, i compilatori tornano alla normale risoluzione degli overload. Per esempio, se sostituiamo il costruttore `std::initializer_list<-bool>` con uno che accetta un `std::initializer_list<std::string>`, i costruttori non `std::initializer_list` tornano a essere selezionabili, poiché non vi è alcun modo per convertire gli int e i bool in `std::string`:

```
class Widget {
public:
  Widget(int i, bool b);          // come prima
  Widget(int i, double d);        // come prima

  // ora il tipo dell'elemento std::initializer_list è std::string
  Widget(std::initializer_list<std::string> il);
  …                               // nessuna funzione
};                                // di conversione implicita

Widget w1(10, true);              // usa le parentesi, ma richiama
                                  il primo costruttore

Widget w2{10, true};              // usa le graffe, ora richiama il
                                  primo costruttore

Widget w3(10, 5.0);               // usa le parentesi, richiama
                                  comunque il secondo costruttore

Widget w4{10, 5.0};               // usa le graffe, ora richiama il
                                  secondo costruttore
```

Questo ci porta verso la fine dell'analisi degli inizializzatori a graffe e

dell’overloading del costruttore, ma vi è ancora un interessante caso limite che occorre considerare. Supponete di utilizzare una coppia di parentesi graffe vuota per costruire un oggetto che supporti la costruzione di default e anche la costruzione `std::initializer_list`. Che significato avranno allora le parentesi graffe vuote? Se significano “nessun argomento”, si ottiene la costruzione di default, ma se significano “`std::initializer_list` vuoto”, si ottiene la costruzione da un `std::initializer_list` senza elementi.

La regola stabilisce che si impieghi la costruzione di default. Le parentesi graffe vuote significano “nessun argomento”, non un `std::initializer_list` vuoto:

```
class Widget {
public:
  Widget();                                   // costr. di default

  Widget(std::initializer_list<int> il);     // costr. std::initializer_list

  …                                           // nessuna funzione
};                                            // di conversione implicita

Widget w1;                                    // richiama il costr. di default

Widget w2{};                                  // richiama il costr. di default

Widget w3();                                  // most vexing parse! dichiara una funzione!
```

Se invece *si intende proprio* richiamare un costruttore `std::initializer_list` con una `std::initializer_list` vuota, occorre rendere le parentesi graffe un argomento del costruttore, inserendo delle parentesi graffe all’interno delle parentesi, o delle parentesi graffe che indicano ciò che si sta passando:

```
Widget w4({});                               // richiama il costr. std::initializer_list
                                             // con una lista vuota
```

```
Widget w5{{}};                    // idem
```

A questo punto, con queste regole apparentemente arcane sugli inizializzatori a graffe, su `std::initializer_list` e sull'overloading del costruttore, potreste chiedervi quanto contino queste informazioni nella programmazione quotidiana. Più di quanto possiate immaginare, poiché una delle classi influenzate direttamente è `std::vector`. `std::vector` ha un costruttore non `std::initializer_list` che consente di specificare le dimensioni iniziali del container e un valore che ognuno degli elementi iniziali dovrà avere; ma ha anche un costruttore che prende un `std::initializer_list` e che permette di specificare i valori iniziali nel container. Se create un `std::vector` di un tipo numerico (per esempio un `std::vector<int>`) e passate al costruttore due argomenti, il fatto che racchiudiate tali argomenti fra parentesi o fra parentesi graffe fa un'enorme differenza:

```
std::vector<int> v1(10, 20);      // usa il costr. non-
                                  std::initializer_list:
                                  // crea un std::vector
                                  // di 10 elementi
                                  // tutti di valore 20


std::vector<int> v2{10, 20};      // usa il costr.
                                  std::initializer_list:
                                  // crea un std::vector di 2
                                  elementi,
                                  // i cui valori sono 10 e 20
```

Ma torniamo un po' indietro rispetto al `std::vector` e anche rispetto ai dettagli delle parentesi, delle graffe e delle regole di risoluzione dell'overloading del costruttore. Da questa discussione derivano due importanti conseguenze. Innanzitutto, come autore di una classe, dovete sapere che se il vostro set di costruttori in overload include una o più funzioni che prendono un `std::initializer_list`, il codice client che utilizza l'inizializzazione a graffe può vedere solo gli overload `std::initializer_list`. Di conseguenza, è

meglio progettare i costruttori in modo che l'overload chiamato non sia influenzato dal fatto che i client utilizzino le parentesi o le graffe. In altre parole, è importante imparare da ciò che ora potete considerare un errore nella progettazione dell'interfaccia di `std::vector` e progettare le classi in modo da scongiurarlo.

Un'implicazione è che se avete una classe senza un costruttore `std::initializer_list` e gliene aggiungete uno, il codice client che utilizza l'inizializzazione a graffe può trovare che le chiamate che un tempo venivano risolte in costruttori non `std::initializer_list`, ora scelgono la nuova funzione. Naturalmente, questo genere di cose può verificarsi ogni volta che si aggiunge una nuova funzione a un set di overload: le chiamate che prima venivano risolte da uno dei vecchi overload possono iniziare a chiamare il nuovo. La differenza con l'overload del costruttore `std::initializer_list` è che un overload `std::initializer_list` non entra mai in competizione con gli altri overload. Li nasconde proprio, fino al punto che gli altri overload non verranno praticamente mai considerati. Dunque, aggiungete tali overload solo con grande cautela.

La seconda lezione è il fatto che, nella classe client, occorre scegliere attentamente fra parentesi e graffe quando si creano gli oggetti. La maggior parte degli sviluppatori sceglie sempre un determinato tipo di delimitatore, utilizzando l'altro solo quando è costretto. Chi preferisce le graffe è attratto dalla loro imbattibile applicabilità, dalla proibizione delle conversioni narrowing e dall'immunità al most vexing parse C++. Questi programmatori sanno che, in alcuni casi (per esempio la creazione di un `std::vector` con determinate dimensioni e un valore iniziale), le parentesi sono necessarie. Al contrario, chi preferisce le normali parentesi impiega le parentesi come delimitatore standard degli argomenti. Si tratta di programmatori attratti dalla loro coerenza con la tradizione sintattica del C++98, dal fatto che evitano il problema dell'**`auto`** che deduce una `std::initializer_list` e dalla sicurezza che le loro chiamate per la creazione di oggetti non richiameranno inavvertitamente costruttori `std::initializer_list`. Ammettono che talvolta solo le parentesi graffe possono risolvere determinate situazioni (per esempio quando si crea un container con particolari valori). Difficile dire che un approccio sia sempre migliore dell'altro e dunque il mio consiglio è quello di sceglierne uno e applicarlo coerentemente.

Se siete autori di template, il "tira-e-molla" fra parentesi e graffe per la creazione

degli oggetti può essere particolarmente frustrante, poiché, in generale, non è possibile sapere quale convenzione sarebbe opportuno usare. Per esempio, supponete di voler creare un oggetto di un tipo arbitrario da un numero arbitrario di argomenti. Un template variadic chiarisce questo concetto:

```
template<typename T, typename... Ts>   // tipo dell'oggetto da creare
                                       // tipi di argomenti da usare
void doSomeWork(Ts&&... params)
{
  crea un oggetto locale T dai parametri...
  …
}
```

Vi sono due modi per trasformare quella riga di solo codice in codice vero (vedi Elemento 25 per informazioni su `std::forward`):

```
T localObject(std::forward<Ts>(params)... );   // parentesi

T localObject{std::forward<Ts>(params)...};    // graffe
```

Considerate, quindi, il seguente codice chiamante:

```
std::vector<int> v;
…
doSomeWork<std::vector<int>>(10, 20);
```

Se `doSomeWork` usa le parentesi quando si crea `localObject`, il risultato è un `std:: vector` di dieci elementi. Se `doSomeWork` usa le parentesi graffe, il risultato è un `std:: vector` di due elementi. Quale dei due risultati è corretto? L'autore di `doSomeWork` non può saperlo. Solo il chiamante lo sa.

Questo è esattamente il problema affrontato dalle funzioni della Libreria Standard `std ::make_unique` e `std::make_shared` (vedi l'Elemento 21). Queste funzioni risolvono il problema utilizzando internamente le parentesi e documentando questa decisione nell'ambito delle loro interfacce.[1]

Argomenti da ricordare

- L’inizializzazione a graffe è la sintassi di inizializzazione utilizzabile più ampiamente: evita le conversioni narrowing ed è immune al most vexing parse del C++.
- Durante la risoluzione dell’overload del costruttore, gli inizializzatori a graffe vengono fatti corrispondere ai parametri `std::initializer_list`, se possibile, anche se altri costruttori offrono corrispondenze apparentemente migliori.
- Un esempio delle situazioni in cui la scelta fra parentesi e graffe può fare una significativa differenza per la creazione di *`std::vector<numeric type>`* con due argomenti.
- La scelta fra parentesi e graffe per la creazione di un oggetto all’interno di un template può essere un problema.

Elemento 8 – Preferire nullptr a 0 e NULL

Ecco il problema: il letterale `0` è un `int` e non un puntatore. Se il C++ trova uno `0` in un contesto in cui può essere utilizzato solo un puntatore, interpreterà questo `0` come un puntatore nullo, ma questa è una posizione di ripiego. La politica principale del C++ è che uno `0` sia un `int` e non un puntatore.

Parlando in senso meno teorico, lo stesso vale per `NULL`. Vi è un po’ di incertezza nei dettagli nel caso di `NULL`, poiché le implementazioni possono dare a `NULL` un tipo intero diverso da `int` (per esempio `long`). La cosa è poco comune, ma questo non conta, poiché qui il problema non è l’esatto tipo di `NULL`, è il fatto che né `0` né `NULL` sono di tipo puntatore.

In C++98, la principale implicazione di ciò era che l’overloading sui tipi puntatore e interi poteva portare a sorprese. Passando `0` o `NULL` a tali overload, non si richiamava mai l’overloading per i puntatori:

```
void f(int);                          // tre overload di f
void f(bool);
void f(void*);
```

```
f(0);                    // richiama f(int), non f(void*)

f(NULL);                 // a volte non compilabile, ma in genere richiama
                         // f(int). Non richiama mai f(void*)
```

L’incertezza relativa al comportamento di `f(NULL)` è un riflesso del margine d’azione concesso alle implementazioni a proposito del tipo di `NULL`. Se `NULL` è definito per essere, per esempio, `0L` (ovvero `0` inteso come `long`), la chiamata è ambigua, poiché le conversioni da `long` a `int`, da `long` a `bool` e da `0L` a `void*` sono considerate ugualmente valide. La cosa interessante su questa chiamata è la contraddizione fra il significato *apparente* del codice sorgente (“sto chiamando `f` con `NULL`, il puntatore nullo”) e il suo significato *effettivo* (“sto chiamando `f` con un certo tipo di intero, non il puntatore nullo”). Questo comportamento anti-intuitivo è ciò che ha portato all’indicazione per i programmatori C++98 di evitare l’overloading sui puntatori e i tipi interi. Tale indicazione rimane valida in C++11, poiché, nonostante l’avviso di questo Elemento, è probabile che alcuni sviluppatori continuino a usare `0` o `NULL` anche nei casi in cui `nullptr` rappresenti la scelta migliore.

Il vantaggio di `nullptr` è che non ha un tipo intero. Per essere onesti, non ha neppure un tipo puntatore, ma lo si può considerare come un puntatore a *tutti i tipi*. L’effettivo tipo di `nullptr` è `std::nullptr_t` e, in una definizione meravigliosamente circolare, `std::nullptr_t` è definito di tipo `nullptr`. Il tipo `std::nullptr_t` si converte implicitamente a tutti i tipi di puntatore standard e questo può far sì che `nullptr` si comporti come se fosse un puntatore di tutti i tipi.

Chiamando la funzione in overload `f` con `nullptr` si richiama l'overload `void*` (ovvero l’overload per puntatori), poiché `nullptr` non può essere visto come qualcosa di intero:

```
f(nullptr);              // richiama l’overload f(void*)
```

Utilizzando `nullptr` invece di `0` o `NULL`, si evitano pertanto sorprese di risoluzione dell’overload, ma questo non è l’unico vantaggio. Migliora anche la chiarezza del codice, specialmente quando vengono impiegate variabili `auto`. Per

esempio, supponete di incontrare le seguenti righe in una base di codice:

```
auto result = findRecord( /* argomenti */ );
if (result == 0) {
…
}
```

Se non sapete (o è difficile capire) ciò che viene restituito da `findRecord`, può non essere chiaro se `result` è un tipo puntatore o un tipo intero. Dopo tutto, `0` (ciò con cui viene confrontato `result`) potrebbe essere entrambe le cose. Se invece vedete il seguente frammento di codice,

```
auto result = findRecord( /* argomenti */ );
if (result == nullptr) {
…
}
```

non vi sono ambiguità: `result` deve essere un tipo puntatore.

`nullptr` è però particolarmente efficace quando entrano in gioco i template. Supponete di avere delle funzioni che dovrebbero essere richiamate solo quando il mutex appropriato è stato bloccato. Ogni funzione accetta un tipo differente di puntatore:

```
int f1(std::shared_ptr<Widget> spw);      // richiama questi
double f2(std::unique_ptr<Widget> upw);   // solo quando è bloccato
bool f3(Widget* pw);                      // il mutex appropriato
```

Il codice chiamante che vuole passare dei puntatori nulli potrebbe avere il seguente aspetto:

```
std::mutex f1m, f2m, f3m;          // mutex per f1, f2 e f3

using MuxGuard =                   // typedef C++11; vedi Elemento 9
  std::lock_guard<std::mutex>;
…
```

```
{
  MuxGuard g(f1m);                // blocca il mutex per f1
  auto result = f1(0);            // passa a f1 0 come ptr null
}                                 // sblocca il mutex

…

{
  MuxGuard g(f2m);                // blocca il mutex per f2
  auto result = f2(NULL);         // passa a f2 NULL come ptr nullo
}                                 // sblocca il mutex

…

{
  MuxGuard g(f3m);                // blocca il mutex per f3
  auto result = f3(nullptr);      // passa a f3 nullptr come ptr
                                  nullo
}                                 // sblocca il mutex
```

L'impossibilità di utilizzare `nullptr` nelle prime due chiamate di questo codice è un vero peccato, ma il codice funziona e questo è importante. Tuttavia, lo schema ripetuto nel codice chiamante (blocco del mutex, chiamata a funzione, sblocco del mutex) non è certo elegante. Questo genere di duplicazione del codice sorgente è una delle cose che possono essere evitate tramite i template, pertanto trasformiamo lo schema in un template:

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
  MuxGuard g(mutex);
  return func(ptr);
```

```
}
```

Se il tipo restituito dalla funzione (auto … -> decltype(func(ptr)) vi rende perplessi, fatevi un favore e leggete l'Elemento 3, che spiega ciò che accade. Qui vedrete che in C++14, il tipo restituito può essere ridotto a un semplice decltype(auto) :

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>

decltype(auto)                              // C++14
lockAndCall(FuncType fune,
                   MuxType&
        mutex,
                   PtrType ptr)
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

Dato il template `lockAndCall` (entrambe le versioni), i chiamanti possono impiegare del codice di chiamata come il seguente:

```
auto result1 = lockAndCall(f1,              // errore!
f1m, 0);
…

auto result2 = lockAndCall(f2,              // errore!
f2m, NULL);
…

auto result3 = lockAndCall(f3,              // corretto
f3m, nullptr);
```

Bene, *possono* scriverlo, ma, come indicano i commenti, in due casi su tre, il codice non verrà compilato. Il problema nella prima chiamata è che quando a `lockAndCall` viene passato `0`, la deduzione del tipo del template non riesce a

determinarne il tipo. Il tipo di `0` è, era e sarà sempre `int` e dunque questo è il tipo del parametro `ptr` all'interno dell'istanziazione di questa chiamata a `lockAndCall`. Sfortunatamente, questo significa che nella chiamata a `func` all'interno di `lockAndCall` viene passato un `int` e questo non è compatibile con il parametro `std::shared_ptr<Widget>` che `f1` si aspetta. Lo `0` passato nella chiamata `lockAndCall` intendeva rappresentare un puntatore nullo, mentre ciò che viene passato è un semplice `int`. Il tentativo di passare questo `int` a `f1` come un `std::shared_ptr<Widget>` è un errore di tipo. La chiamata a `lockAndCall` con `0` è errata, poiché all'interno del template viene passato un `int` a una funzione che richiede un `std::shared_ptr<Widget>`.

L'analisi della chiamata che riguarda `NULL` è sostanzialmente la stessa. Quando a `lockAndCall` viene passato `NULL`, viene dedotto un tipo intero per il parametro `ptr` e si verifica un errore di tipo quando `ptr` (un `int` o un tipo analogo) viene passato a `f2`, la quale si aspetta un `std::unique_ptr<Widget>`.

Al contrario, la chiamata che usa `nullptr` non ha problemi. Quando a `lockAndCall` si passa `nullptr`, per `ptr` viene dedotto il tipo `std::nullptr_t`. Quando `ptr` viene passato a `f3`, esiste una conversione implicita da `std::nullptr_t` a `Widget*`, poiché `std::nullptr_t` si converte implicitamente in tutti i tipi puntatore.

Il fatto che la deduzione del tipo per il template deduce il tipo "sbagliato" per `0` e `NULL` (ovvero i loro veri tipi invece che la loro rappresentazione sotto forma di puntatore nullo) è il motivo principale che spinge a utilizzare `nullptr` al posto di `0` o `NULL` quando si vuole fare riferimento a un puntatore nullo. Con `nullptr`, i template non fanno alcuna obiezione. Combinati con il fatto che `nullptr` non soffre delle sorprese di risoluzione dell'overload cui sono suscettibili `0` e `NULL`, in pratica non vi sono alternative: quando si vuole fare riferimento a un puntatore nullo, utilizzate `nullptr` e non `0` o `NULL`.

Argomenti da ricordare

- Preferite `nullptr` a `0` e `NULL`.
- Evitate l'overloading dei tipi interi e puntatore.

Elemento 9 – Preferire le dichiarazioni alias ai typedef

Penso che siamo tutti d'accordo che l'utilizzo di container STL è una buona idea e spero che l'Elemento 18 vi convinca che anche utilizzare `std::unique_ptr` è una buona idea, ma sospetto che nessuno ami scrivere cose come "`std::unique_ptr<std::unordered_map<std::string, std::string>>`" più di una volta. Anche il solo pensarci sembra aumentare il rischio di sindrome del tunnel carpale.

Evitare questo tipo di danni fisici è facile. Basta usare un `typedef`:

```
typedef
    std::unique_ptr<std::unordered_map<std::string, std::string>>
    UPtrMapSS;
```

Ma i `typedef` son decisamente "troppo C++98". Funzionano, sì, anche in C++11, che però offre anche le *dichiarazioni alias:*

```
using UPtrMapSS =
    std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

Dato che i `typedef` e le dichiarazioni alias fanno esattamente la stessa cosa, è ragionevole chiedersi se esista un solido motivo tecnico per preferire una forma rispetto all'altra.

In realtà c'è, ma prima di parlarne, occorre dire che sono in molti a trovare la dichiarazione alias più facile da gestire quando si ha a che fare con tipi che prevedono puntatori a funzione:

```
// FP è sinonimo di puntatore a una funzione che accetta un int e
// un const std::string& e non restituisce nulla returning nothing
typedef void (*FP)(int, const std::string&);     // typedef

// stesso significato della forma precedente
using FP = void (*)(int, const std::string&);    // dichiarazione
                                                 // alias
```

Nessuna delle due forme è particolarmente comoda e solo pochi dedicheranno del tempo a preoccuparsi dei sinonimi per i tipi puntatori a funzione, pertanto questa non può essere considerata una motivazione molto convincente per spingere a preferire le dichiarazioni alias ai `typedef`.

Ma una motivazione convincente esiste eccome: i template. In particolare, le dichiarazioni alias possono essere rese template (nel qual caso sono chiamate *template alias*), mentre ciò non è possibile con i `typedef`. Ciò dà ai programmatori C++11 un meccanismo diretto per esprimere cose che in C++98 dovevano essere risolte con dei `typedef` nidificati all'interno di `struct` templatizzate. Per esempio, considerate la definizione di un sinonimo per una lista concatenata che utilizza un allocatore personalizzato, `MyAlloc`. Con un template alias, è semplicissimo:

```
template<typename T>                   // MyAllocList<T>
using MyAllocList = std::list<T,       // è sinonimo di
MyAlloc<T>>;
                                       // std::list<T,
                                       // MyAlloc<T>>

MyAllocList<Widget> lw;                // codice client
```

Con un `typedef`, si deve praticamente creare tutto dall'inizio:

```
template<typename T>                   // MyAllocList<T>::type
struct MyAllocList {                   // è sinonimo di
  typedef std::list<T,
  MyAlloc<T>> type;                    // std::list<T,
};                                     // MyAlloc<T>>

MyAllocList<Widget>::type lw;          // codice client
```

Ma c'è di peggio. Se si vuole utilizzare il `typedef` all'interno di un template, con lo scopo di creare una lista concatenata di oggetti di un tipo specificato da un parametro del template, occorre far precedere `typename` al nome del `typedef`:

```
template<typename T>
```

```
class Widget {                          // Widget<T> contiene
private:                                // un MyAllocList<T>
  typename MyAllocList<T>::type
  list;                                 // dato membro
  …
};
```

Qui, `MyAllocList<T>::type` fa riferimento a un tipo che dipende da un parametro del template (**`T`**). `MyAllocList<T>::type` è pertanto un *tipo dipendente* e una delle regole più interessanti del C++ è che i nomi dei tipi dipendenti devono essere preceduti da `typename`.

Se `MyAllocList` è definito come un template alias, questa necessità di usare `typename` sparisce (e anche il macchinoso suffisso “`::type`”):

```
template<typename T>
using MyAllocList = std::list<T,
MyAlloc<T>>;                            // come prima

template<typename T>
class Widget {
private:
  MyAllocList<T> list;                  // niente "typename",
  …                                     // niente "::type"
};
```

Per voi, `MyAllocList<T>` (ovvero l’uso del template alias) può sembrare altrettanto dipendente dal parametro template `T` di `MyAllocList<T>::type` (ovvero l’uso del `typedef` nidificato), ma il compilatore la vede diversamente. Quando i compilatori elaborano il template di `Widget` e incontrano il `MyAllocList<T>` (ovvero il template alias), sanno che `MyAllocList<T>` è il nome di un tipo: poiché `MyAllocList` è un template alias, *deve* essere il nome di un tipo. `MyAllocList` è pertanto un *tipo non dipendente* e dunque lo specificatore `typename` non è né necessario né permesso.

Quando invece i compilatori vedono `MyAllocList<T>::type` (ovvero l’uso del `typedef` nidificato) nel template `Widget`, non possono essere certi che esso indichi un tipo, poiché vi potrebbe essere una specializzazione di `MyAllocList`

che non hanno ancora visto, in cui `MyAllocList<T>::type` fa riferimento a qualcosa di diverso da un tipo. Sembra assurdo, ma non è certo colpa dei compilatori se questa possibilità esiste. Semmai sono gli esseri umani che possono produrre codice di questo tipo.

Per esempio, qualche anima perduta potrebbe aver concepito qualcosa di simile a:

```
class Wine { … };

template<>                          // specializzazione di MyAllocList
class MyAllocList<Wine> {           // per T uguale a Wine
private:
  enum class WineType               // vedi Elemento 10 per info su
  { White, Red, Rose };             // "enum class"

  WineType type;                    // in questa classe, type è
  …                                 // un dato membro!
};
```

Come potete vedere, `MyAllocList<Wine>::type` non fa riferimento a un tipo. Se `Widget` dovesse essere istanziato con `Wine`, il `MyAllocList<Wine>::type` all'interno del template di `Widget` farebbe riferimento a un dato membro e non a un tipo. All'interno del template di `Widget`, quindi, il fatto che `MyAllocList<Wine>::type` faccia riferimento a un tipo dipende da che cos'è `T` e questo è il motivo per cui i compilatori insistono sul fatto che specifichiate esplicitamente che questo è un tipo, facendogli precedere `typename`.

Se avete svolto metaprogrammazione con template, quasi certamente vi sarete scontrati con la necessità di prendere i parametri di tipo del template e creare, partendo da loro, dei tipi riveduti. Per esempio, dato un certo tipo `T`, potreste volere eliminare eventuali qualificatori `const` o di riferimento contenuti in `T`; per esempio, potreste voler trasformare una `const std::string&` in una `std::string`. Oppure potreste voler aggiungere `const` a un tipo o trasformarlo in un riferimento lvalue, ovvero trasformare un `Widget` in un `const Widget` o in un `Widget&`. Se non vi occupate di metaprogrammazione con template, è un peccato, poiché se volete essere davvero efficaci come programmatori C++,

dovete conoscere almeno le basi di questo aspetto del linguaggio. Potete vedere in azione degli esempi di metaprogrammazione con template, comprese le trasformazioni di tipo appena menzionate, nei Punti 23 e 27.

Il C++11 offre degli strumenti per svolgere questo genere di trasformazioni, sotto forma di *type trait*, un assortimento di template all'interno dell'header `<type_traits>`. Esistono decine di type trait in tale header e non tutti svolgono trasformazioni di tipo, ma quelli che lo fanno offrono un'interfaccia prevedibile. Dato un tipo `T` al quale vorreste applicare una trasformazione, il tipo risultante è `std::`*`transformazione`*`<T>::type`. Ecco un esempio:

```
std::remove_const<T>::type          // fornisce T da const T

std::remove_reference<T>::type      // fornisce T da T& e T&&

std::add_lvalue_reference<T>::type // fornisce T& da T
```

I commenti riepilogano semplicemente ciò che fanno queste trasformazioni, pertanto non prendeteli troppo alla lettera. Prima di utilizzarli in un progetto, è opportuno considerare le esatte specifiche.

La mia motivazione, qui, non è quella di offrirvi una guida ai type trait. Piuttosto, notate che l'applicazione di queste trasformazioni prevede la scrittura di "**`::type`**" alla fine di ciascun utilizzo. Se li applicate a un parametro di tipo all'interno di un template (che è sostanzialmente il modo in cui li impiegate nel codice vero e proprio), dovrete anche far precedere ciascun uso con `typename`. Il motivo di entrambi questi intoppi sintattici è il fatto che i type trait del C++11 sono implementati come `typedef` nidificati all'interno di `struct` templatizzate. Esattamente: sono implementate utilizzando la tecnologia del sinonimo del tipo, mentre ho appena tentato di convincervi che è inferiore ai template alias!

Esiste una motivazione storica per questa scelta, ma non è il caso di parlarne, poiché il comitato di standardizzazione ha riconosciuto che i template alias sono il miglior modo per procedere e hanno incluso tali template in C++14 per tutte le trasformazioni di tipo C++11. Gli alias hanno una forma comune: per ogni trasformazione C++11 `std::`*`trasformazione`*`<T>::type`, esiste un corrispondente template alias C++14 chiamato `std::`*`trasformazione`*`_`**`t`**. I seguenti esempi chiariscono questo fatto:

```
std::remove_const<T>::type          // C++11: const T → T
std::remove_const_t<T>              // equivalente in C++14

std::remove_reference<T>::type      // C++11: T&/T&& → T
std::remove_reference_t<T>          // equivalente in C++14

std::add_lvalue_reference<T>::type  // C++11: T → T&
std::add_lvalue_reference_t<T>      // equivalente in C++14
```

I costrutti C++11 rimangono validi anche in C++14, ma, in realtà, è difficile capire per quale motivo dovreste usarli. Anche se non potete impiegare il C++14, la scrittura dei template alias è un gioco da ragazzi. Banalmente, basta usare le funzionalità del linguaggio C++11. Se avete accesso a una copia elettronica dello standard C++14, è ancora più facile, poiché tutto ciò che è necessario è un po' di copia-e-incolla. Cominciamo da qui:

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t =
  typename add_lvalue_reference<T>::type;
```

Visto? Non potrebbe essere più facile.

Argomenti da ricordare

- I `typedef` non supportano la templatizzazione, mentre le dichiarazioni alias sì.
- I template alias evitano di utilizzare il suffisso `::type` e, nei template, il prefisso "`typename`", spesso necessario per far riferimento ai `::type`.
- Il C++14 offre i template alias per tutte le trasformazioni type trait del C++11.

Elemento 10 – Preferire gli enum con visibilità a quelli senza visibilità

Come regola generale, la dichiarazione di un nome fra parentesi graffe limita la visibilità (*scope*) di tale nome all'interno di tali graffe. Questo non si applica agli enumeratori dichiarati negli `enum` in stile C++98. I nomi di tali enumeratori appartengono al campo di visibilità contenente l'`enum`; ciò significa che niente altro nello stesso campo di visibilità può avere lo stesso nome:

```
enum Color { black, white, red }; // black, white, red sono
                                  // nello stesso campo di
                                  visibilità di Color

auto white = false;               // errore! white già
                                  // dichiarato in questo scope
```

Il fatto che questi nomi fuoriescano dall'enum per adottare il campo di visibilità che contiene la definizione della loro enum dà origine al termine ufficiale per questo genere di enum: *senza campo di visibilità (unscoped).* La loro nuova versione C++11, le enum *con campo di visibilità (scoped),* non soffre di questo problema:

```
enum class Color { black, white,  // black, white, red
red };
                                  // si limitano a Color

auto white = false;               // OK, nessun altro
                                  // "white" nello scope

Color c = white;                  // errore! Non esiste un
                                  enumeratore
                                  // chiamato “white” in questo
                                  scope
Color c = Color::white;           // OK

auto c = Color::white;            // OK (e concorde
```

```
                                  // alle indicazioni dell'Elemento
                                  5)
```

Poiché le enum con campo di visibilità sono dichiarate tramite “`enum class`”, vengono talvolta chiamate *classi* `enum`.

La riduzione dell’inquinamento dello spazio dei nomi offerta dalle `enum` con campo di visibilità è motivo sufficiente a preferirle rispetto alla versione senza campo di visibilità, ma le prime hanno anche un altro grande vantaggio: i loro enumeratori hanno una tipizzazione più forte. Gli enumeratori delle `enum` senza campo di visibilità si convertono implicitamente in tipi interi (e, da qui, in tipi in virgola mobile). Pertanto, sono perfettamente valide le seguenti acrobazie semantiche:

```
enum Color { black, white, red }; // enum unscoped

std::vector<std::size_t>          // funz. Che restituisce
  primeFactors(std::size_t x);    // i fattori primi di x

Color c = red;
…

if (c < 14.5){                    // confronta Color con double (!)
  auto factors =                  // calcola i fattori
  primeFactors(c);                // primi di un Color (!)
  …
}
```

Mettete un semplice “`class`” dopo “`enum`”, trasformando così una `enum` senza campo di visibilità in una con campo di visibilità, e tutto cambia. Non esistono conversioni implicite dagli enumeratori di una `enum` con campo di visibilità verso qualsiasi altro tipo:

```
enum class Color { black, white,
red };                            // ora enum è scoped

Color c = Color::red;             // come prima, ma
                                  // con qualificatore di scope
```

```
if (c < 14.5) {                  // errore! Impossibile confrontare
                                 // Color e un double
  auto factors =                 // errore! Impossibile passare Color a una
    primeFactors(c);             // funzione che si aspetta un std::size_t
  …
}
```

Se davvero volete eseguire una conversione da `Color` a un altro tipo, fate ciò che avete sempre fatto per cambiare il tipo in ciò che desiderate: utilizzate una cast.

```
if (static_cast<double>(c) < 14.5) {   // non bellissimo,
                                       // ma valido
auto factors =                         // sospetto,
  primeFactors(static_cast<std::size_t>(c));  // ma compilabile
…
}
```

Può sembrare che le `enum` con campo di visibilità abbiano un terzo vantaggio rispetto a quelle senza campo di visibilità, poiché le prime possono essere dichiarate in forward, ovvero il loro nome può essere dichiarato senza specificare gli enumeratori:

```
enum Color;                      // errore!
enum class Color;                // OK
```

Questo è fuorviante. In C++11, anche le `enum` senza campo di visibilità possono essere dichiarate in forward, ma solo dopo un po' di lavoro aggiuntivo. Tale lavoro deriva dal fatto che in C++ ogni `enum` ha un *tipo intero base* determinato dai compilatori. Per una `enum` senza campo di visibilità come `Color`,

```
enum Color { black, white, red };
```

i compilatori devono scegliere `char` come tipo base, poiché vi sono solo tre

valori da rappresentare. Tuttavia, alcune `enum` hanno un intervallo di valori molto più esteso, per esempio:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

Qui i valori da rappresentare vanno da `0` a `0xFFFFFFFF`. Tranne su macchine molto particolari (dove un `char` sia costituito da almeno 32 bit), i compilatori dovranno selezionare un tipo intero di dimensioni maggiori di `char` per rappresentare i valori di `Status`.

Per fare un utilizzo efficiente della memoria, spesso i compilatori scelgono il tipo base più piccolo per una `enum`, ma che sia sufficiente a rappresentare la gamma dei valori enumerativi. In alcuni casi, però, i compilatori eseguiranno l'ottimizzazione favorendo la velocità, invece delle dimensioni; in tal caso potrebbero non scegliere il tipo base più piccolo, ma certamente vorranno avere la possibilità di eseguire una certa ottimizzazione delle dimensioni. Per rendere possibile tutto questo, il C++98 supporta solo le definizioni `enum` (dove sono elencati tutti gli enumeratori); le dichiarazioni `enum` non sono permesse. Ciò consente ai compilatori di selezionare un tipo base per ciascuna `enum` prima che la `enum` stessa venga utilizzata.

Ma l'impossibilità di dichiarare in forward le `enum` ha dei difetti. Il principale è probabilmente l'aumento delle dipendenze in fase di compilazione. Considerate nuovamente l'`enum Status`:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

Questo è un genere di `enum` che verrà probabilmente utilizzata in più punti del sistema, pertanto verrà inclusa in un file header da cui dipenderà ogni parte del sistema. Se in `Status` venisse introdotto un nuovo valore,

```
enum Status { good = 0,
               failed = 1,
               incomplete = 100,
               corrupt = 200,
               audited = 500,
               indeterminate = 0xFFFFFFFF
             };
```

è probabile che l'intero sistema debba essere ricompilato, e questo anche se l'enumerazione è usata da un unico sottosistema (magari da un'unica funzione!). Questo è il genere di cose che i programmatori *detestano*. E questo è il genere di cose che viene eliminato dalla possibilità di dichiarare in forward le `enum` in C++11. Per esempio, ecco una dichiarazione perfettamente valida di una `enum` con campo di visibilità e di una funzione che la accetta come parametro:

```
enum class Status;                  // dichiarazione forward

void continueProcessing(Status      // uso dell'enum dichiarata
s);                                 forward
```

L'header contenente queste dichiarazioni non richiede alcuna ricompilazione qualora venisse modificata la definizione di `Status`. Inoltre, se `Status` viene modificata (per esempio aggiungendo l'enumeratore `audited`), ma il comportamento di `continueProcessing` non cambia (per esempio perché `continueProcessing` non utilizza `audited`), neppure l'implementazione di `continueProcessing` dovrà essere ricompilata.

Ma se il compilatore deve conoscere le dimensioni di una `enum` prima che venga utilizzata, come possono le `enum` del C++11 impiegare le dichiarazioni forward laddove quelle del C++98 non possono? La risposta è semplice: il tipo base di un**'enum** con campo di visibilità è sempre noto, mentre per le `enum` senza campo di visibilità deve essere specificato.

Per default, il tipo base per le `enum` con campo di visibilità è `int`:

```
enum class Status;                  // il tipo base è int
```

Se il caso di default non è adatto, lo si può modificare:

```
                                    // il tipo base per
```

```
enum class Status: std::uint32_t;
                                  // Status è std::uint32_t
                                  // (da <cstdint>)
```

In entrambi i casi, il compilatore conosce le dimensioni degli enumeratori di una `enum` con campo di visibilità.

Per specificare il tipo base di una `enum` senza campo di visibilità, si fa la stessa cosa vista per una con campo di visibilità e il risultato può essere dichiarato in forward:

```
enum Color: std::uint8_t;         // dich. fwd per enum unscoped;
                                  // il tipo base è
                                  // std::uint8_t
```

Le specifiche del tipo base possono andare anche nella definizione di una enum:

```
enum class Status: std::uint32_t
                    { good = 0,
                      failed = 1,
                      incomplete = 100,
                      corrupt = 200,
                      audited = 500,
                      indeterminate = 0xFFFFFFFF
                    };
```

Dato il fatto che le `enum` con campo di visibilità evitano l'inquinamento dello spazio dei nomi e non sono suscettibili a conversioni di tipo implicite insensate, può essere sorprendente sapere che vi è almeno una situazione in cui possono invece essere utili le `enum` senza campo di visibilità. Capita quando si fa riferimento a campi all'interno delle `std::tuple` del C++11. Per esempio, supponete di avere una tupla che contiene valori per il nome, l'indirizzo di posta elettronica e il valore di reputazione di un utente di un social network:

```
using UserInfo =                  // tipo alias; vedi Elemento 9
  std::tuple<std::string,         // nome
  std::string,                    // email
```

```
  std::size_t> ;                  // reputazione
```

Anche se i commenti indicano cosa rappresenta ciascun campo della tupla, questo probabilmente non è molto utile quando si incontra codice come il seguente in un file di codice sorgente distinto:

```
UserInfo uInfo;                  // oggetto di tipo tupla
…
auto val = std::get<1>(uInfo);   // prendi il valore del campo 1
```

Come programmatori, avrete molte cose di cui preoccuparvi. Davvero pensate di ricordarvi che il campo 1 corrisponde all'indirizzo di posta elettronica dell'utente? Probabilmente no. Utilizzando una `enum` senza campo di visibilità per associare i nomi ai numeri dei campi si evita questa necessità:

```
enum UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                  // come prima
…
auto val = std::get<uiEmail>
(uInfo);                         // ah, è il valore del
                                 // campo per l'email
```

Ciò che lo fa funzionare è la conversione implicita da `UserInfoFields` a `std::size_t`, che è il tipo richiesto da `std::get`.

Il codice corrispondente con le `enum` con campo di visibilità è notevolmente più complesso:

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                  // come prima
…

auto val =
  std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
    (uInfo);
```

Tale complessità può essere ridotta scrivendo una funzione che prende un enumeratore e restituisce il suo corrispondente valore `std::size_t`, ma la cosa diventa macchinosa. La `std::get` è un template e il valore fornito è un argomento template (notate infatti l'uso delle parentesi angolari e non delle parentesi) e, pertanto, la funzione che trasforma un enumeratore in un `std::size_t` deve produrre il proprio risultato *durante la compilazione*. Come spiega l'Elemento 15, ciò significa che deve essere una funzione `constexpr`.

In realtà, dovrebbe essere un template di funzione `constexpr`, poiché dovrebbe lavorare con ogni genere di `enum`. E se bisogna fare tale generalizzazione, occorre generalizzare anche il tipo restituito. Invece di restituire `std::size_t`, si restituirà il tipo base dell'**`enum`.** Questo è disponibile tramite `std::underlying_type` (per informazioni consultate l'Elemento 9). Infine, lo si dichiara `noexcept` (vedi Elemento 14) poiché sappiamo che non lancerà mai un'eccezione. Il risultato è una funzione template `toUType` che prende un enumeratore arbitrario e può restituire il suo valore come una costante nota in fase di compilazione:

```
template<typename E>
constexpr typename std::underlying_type<E>::type
         toUType(E enumerator) noexcept
{
         return
               static_cast<typename
          std::underlying_type<E>::type>(enumerator);
}
```

In C++14, `toUType` può essere semplificato sostituendo `typename std::underlying_type<E>::type` con `std::underlying_type_t` (vedi Elemento 9):

```
template<typename E>      // C++14
constexpr std::underlying_type_t<E>
    toUType(E enumerator) noexcept
{
    return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Ancora più compatto è il tipo restituito con `auto` (vedi Elemento 3) valido anch'esso in C++14:

```
template<typename E>      // C++14
```

```
constexpr auto
    toUType(E enumerator) noexcept
{
    return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Indipendentemente dal modo in cui è scritto, `toUType` ci permette di accedere a un campo della tupla nel seguente modo:

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

Sempre più testo da scrivere rispetto a una `enum` senza visibilità, ma in più evita l'inquinamento dello spazio dei nomi e le conversioni incontrollate che riguardano gli enumeratori. In molti casi si può decidere che qualche carattere in più da scrivere sia un prezzo ragionevole da pagare per la possibilità di evitare le trappole di una tecnologia per enumerazioni "preistorica", dato che risale ai tempi in cui le comunicazioni digitali avvenivano tramite modem a 2400 baud.

Argomenti da ricordare

- Le `enum` in stile C++98 sono ora chiamate `enum` senza campo di visibilità, *unscoped*.
- Gli enumeratori delle `enum` con campo di visibilità *(scoped)* sono visibili solo all'interno della `enum`. Si convertono in altri tipi solo con un cast.
- Le `enum` con e senza campo di visibilità supportano la specifica del tipo base. Per default il tipo base delle `enum` con campo di visibilità è `int`, mentre le `enum` senza campo di visibilità non hanno alcun default che determina il tipo base.
- Le `enum` con campo di visibilità possono sempre essere dichiarate in forward. Quelle senza campo di visibilità possono essere dichiarate in forward solo se la loro dichiarazione specifica un tipo base.

Elemento 11 – Preferire le funzioni "cancellate" a quelle private undefined

Se dovete fornire del codice ad altri sviluppatori e volete impedire loro di richiamare una determinata funzione, generalmente non dovete dichiarare tale funzione. Nessuna dichiarazione di funzione, nessuna funzione da richiamare. Assolutamente semplice. Ma talvolta il linguaggio C++ dichiara delle funzioni per voi e se volete impedire che i client possano richiamare tali funzioni, le cose si fanno un po' più complesse.

La situazione sorge solo per le "funzioni membro speciali", ovvero le funzioni membro che il C++ genera automaticamente in caso di necessità. L'Elemento 17 tratta in dettaglio queste funzioni, ma, per il momento, ci occuperemo solo del costruttore per copia e dell'operatore di assegnamento per copia. Questo capitolo è ampiamente dedicato a quelle pratiche che, comuni in C++98, sono state superate da tecniche migliori introdotte dal C++11; e in C++98, se volete sopprimere l'uso di una funzione membro, si tratta quasi sempre di un costruttore per copia, dell'operatore di assegnamento per copia o di entrambe le cose.

L'approccio C++98 per impedire l'uso di queste funzioni consiste nel dichiararle come `private` e non definirle. Per esempio, verso la base della gerarchia `iostream` nella Libreria Standard C++ si trova la classe template `basic_ios`. Tutte le classi `istream` e `ostream` ereditano (magari indirettamente) da questa classe. Copiare `istream` e `ostream` è indesiderabile, poiché non è chiaro completamente che cosa devono fare tali operazioni. Un oggetto `istream`, per esempio, rappresenta uno stream di valori di input, alcuni dei quali possono essere già stati letti e alcuni dei quali verranno potenzialmente letti in un secondo tempo. Se un `istream` dovesse essere copiato, ciò prevedrà la copia di tutti i valori che sono già stati letti e anche dei valori che verranno letti in futuro? Il modo più semplice per gestire una situazione di questo tipo consiste nel definirli fuori dall'esistenza. Proibire la copia degli stream fa proprio questo.

Per rendere incopiabili le classi `istream` e `ostream`, `basic_ios` è specificato in C++98 nel seguente modo (vedere i commenti):

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  …

private:
  basic_ios(const basic_ios& );   // non definito
```

```
  basic_ios& operator=(const
  basic_ios&);                    // non definito
};
```

Dichiarando queste funzioni come `private` si evita che i client possano richiamarle. Il fatto di non definirle deliberatamente significa che se il codice che può comunque accedere a queste funzioni (per esempio le funzioni membro o le funzioni `friend` della classe) le usa, il linking non riuscirà a causa della mancanza della definizione delle funzioni.

In C++11 vi è un modo migliore per ottenere essenzialmente la stessa cosa: usare "`= delete`" per contrassegnare il costruttore per copia e l'operatore di assegnamento per copia come *funzioni cancellate*. Ecco la stessa parte di `basic_ios` specificata in C++11:

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  …
  basic_ios(const basic_ios& ) = delete;
  basic_ios& operator=(const basic_ios&) = delete;
  …
};
```

La differenza fra cancellare queste funzioni e dichiararle `private` può sembrare puramente estetica, ma, in realtà, vi è molta più sostanza di quanto si possa immaginare. Le funzioni cancellate non possono essere utilizzate in alcun modo e pertanto anche il codice contenuto nelle funzioni membro e `friend` non verrà compilato se dovesse tentare di copiare oggetti `basic_ios`. Questo rappresenta un miglioramento rispetto al comportamento del C++98, dove tale uso improprio non sarebbe stato diagnosticato se non in fase di linking.

Per convenzione, le funzioni cancellate sono dichiarate `public`, non `private`. Il motivo è il seguente. Quando il codice client cerca di utilizzare una funzione membro, il C++ controlla l'accessibilità prima dello status di "cancellata". Quando il codice client tenta di utilizzare una funzione `private` cancellata, alcuni compilatori si lamentano solo del fatto che la funzione è `private`, anche se l'accessibilità della funzione non influisce sul fatto che essa possa essere utilizzata. È importante tenere a mente questo fatto quando occorre rielaborare del codice non recente per sostituire le funzioni membro `private`e-non-definite con funzioni "cancellate", poiché il fatto di rendere le nuove funzioni `public`

produce, generalmente, messaggi d’errore più “parlanti”.

Un vantaggio importante di questa tecnica è il fatto che può essere cancellata *qualsiasi* funzione, mentre possono essere rese private solo le funzioni membro. Per esempio, supponete di avere una funzione non membro che accetta un intero e restituisce il fatto che tale intero sia un numero fortunato:

```
bool isLucky(int number);
```

Il fatto che il C++ derivi dal C significa che quasi ogni tipo che può essere considerato, anche vagamente, numerico, verrà convertito in `int`, ma alcune chiamate accettate dal compilatore potrebbero non avere senso:

```
if (isLucky('a')) …              // 'a' è un numero fortunato?

if (isLucky(true)) …             // e "true"?

if (isLucky(3.5)) …              // non dovremmo troncarlo a 3
                                 // prima di verificare se è
                                 fortunato?
```

Se i numeri fortunati devono davvero essere interi, vorremmo impedire che chiamate come le precedenti risultino compilabili.

Un modo per ottenerlo consiste nel creare overload cancellati per i tipi che si vogliono eliminare:

```
bool isLucky(int number);        // funzione originale

bool isLucky(char) = delete;     // rifiuta i char

bool isLucky(bool) = delete;     // rifiuta i bool

bool isLucky(double) = delete;   // rifiuta i double
                                 // e i float
```

Il commento sull’overload per i `double` dice che verranno rifiutati sia i `double` sia i `float` potrebbe sorprendere, ma ricordatevi che, data una scelta fra conversione di un `float` in un `int` o in un `double`, il C++ preferisce la

conversione in `double`. Una chiamata a `isLucky` con un `float` richiamerà pertanto l'overload per i `double`, non quello per gli `int`. Almeno cercherà. Il fatto che l'overload sia cancellato bloccherà la chiamata già dalla compilazione.

Anche se le funzioni cancellate non possono essere utilizzate, fanno comunque parte del programma. Pertanto, vengono tenute in considerazione durante la risoluzione degli overload. Questo è il motivo per cui, con le dichiarazioni di funzioni cancellate precedenti, le chiamate indesiderabili a `isLucky` verranno rifiutate:

```
if (isLucky('a')) …            // errore! Chiamata di una
                               funzione cancellata

if (isLucky(true)) …           // errore!


if (isLucky(3.5f)) …           // errore!
```

Un altro trucco che le funzioni cancellate possono impiegare (e le funzioni membro `private` no) consiste nell'impedire l'uso delle istanziazioni del template che dovrebbero essere disabilitate. Per esempio, supponete di aver bisogno di un template che funzioni con i puntatori standard (nel Capitolo 4 si consiglia di preferire i puntatori smart ai semplici puntatori):

```
template<typename T>
void processPointer(T* ptr);
```

Vi sono due casi speciali nel mondo dei puntatori. Uno è rappresentato dai puntatori `void*`, poiché non vi è alcun modo per de-referenziarli, di incrementarli/decrementarli e così via. L'altro è rappresentato dai puntatori `char*`, poiché normalmente rappresentano puntatori a stringhe in stile C, non puntatori a singoli caratteri. Questi casi speciali spesso richiedono una gestione altrettanto speciale e, nel caso del template `processPointer`, supponiamo che la gestione corretta consista nel rifiutare le chiamate che utilizzano questi tipi. In pratica non dovrebbe essere possibile richiamare `processPointer` con un puntatore `void*` o `char*`.

Questo è facile da realizzare, basta cancellare le seguenti istanziazioni:

```
template<>
```

```
void processPointer<void>(void*) = delete;

template<>
void processPointer<char>(char*) = delete;
```

Ora, se non sono valide le chiamate a `processPointer` con `void*` o `char*`, probabilmente non sono valide neanche le chiamate con `const void*` o `const char*` e dunque anche tali istanziazioni dovranno essere cancellate:

```
template<>
void processPointer<const void>(const void*) = delete;

template<>
void processPointer<const char>(const char*) = delete;
```

E se volete davvero essere perfezionisti, dovreste anche cancellare gli overload `const volatile void*` e `const volatile char*` e poi vi dovrete occupare degli overload per i puntatori ad altri tipi di caratteri standard: `std::wchar_t`, `std::char16_t` e `std::char32_t`.

È interessante notare che se avete il template di una funzione all'interno della classe e intendete disabilitare alcune sue istanziazioni dichiarandole `private` (nel modo classico C++98) questo non è possibile, poiché non è possibile dare a una specializzazione del template di una funzione membro un livello d'accesso differente rispetto a quello del template principale. Se `processPointer` fosse il template di una funzione membro all'interno di `Widget`, per esempio, e voleste disabilitare le chiamate a puntatori `void*`, quello che segue sarebbe l'approccio C++98, che però non verrebbe compilato:

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }

private:
  template<>                      // errore!
  void processPointer<void>
```

```
  (void*);

};
```

Il problema è che le specializzazioni del template devono essere scritte con campo di visibilità namespace e non solo della classe. Questo problema non sorge con le funzioni cancellate, che non hanno bisogno di un livello d'accesso differente. Possono essere cancellate all'esterno della classe (da qui il campo di visibilità namespace):

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }
   ...
};

template<>                                        // still
  void Widget::processPointer<void>               // pubblica,
  (void*) = delete;                               // ma cancellata
```

La verità è che la pratica C++98 di dichiarare le funzioni private e poi non definirle era in realtà un tentativo di ottenere ciò che viene realizzato con le funzioni cancellate del C++11. Come emulazione, l'approccio C++98 non è altrettanto efficace. Non funziona all'esterno delle classi, non funziona sempre all'interno delle classi e quando funziona potrebbe non funzionare se non al momento del linking. Pertanto, molto meglio impiegare le funzioni cancellate.

Argomenti da ricordare

- Preferire le funzioni cancellate a quelle private non definite.
- Ogni funzione può essere cancellata, comprese le funzioni non membro e le istanziazioni template.

Elemento 12 – Dichiarare con override le funzioni che lo richiedono

Il mondo della programmazione a oggetti del C++ ruota attorno alle classi, all'ereditarietà e alle funzioni virtuali. Fra le idee su cui si basa questo "mondo" vi è il fatto che le implementazioni delle funzioni virtuali nelle classi derivate hanno la precedenza *(override)* sulle implementazioni nelle relative classi base. È sconfortante, pertanto, scoprire con quale facilità l'overriding delle funzioni virtuali possa dare problemi. È come se questa parte del linguaggio fosse stata progettata con l'idea che la nota Legge di Murphy non solo fosse valida, ma dovesse essere rispettata.

Poiché la parola "overriding" suona un po' come "overloading", anche se si tratta di concetti completamente distinti, chiariamo che l'overriding di funzioni virtuali è ciò che rende possibile richiamare una funzione in una classe derivata tramite l'interfaccia della classe base:

```
class Base {
public:
  virtual void doWork();    // funzione virtuale della classe base
  …
};

class Derived: public Base
{
public:
  virtual void doWork();    // override di Base::doWork
  …                         // (qui "virtual"
}; // è facoltativo)

std::unique_ptr<Base> upb =  // crea un puntatore della classe base
  std::make_unique<Derived>(); // che punta a un oggetto della classe derivata;
                             // vedi Elemento 21 per info su
```

```
…                                   // std::make_unique
upb->doWork();                      // richiama doWork tramite il puntatore
                                    // alla classe base; richiama la funzione
                                    // nella classe derivata
```

Perché possa verificarsi l'overriding, devono valere alcune condizioni:

- la funzione nella classe base deve essere virtuale;
- il nome della funzione base e derivata deve essere lo stesso (tranne nel caso dei distruttori).
- I tipi di parametri delle funzioni base e derivata devono essere identici;
- Il modificatore `const` nelle funzioni base e derivata deve essere lo stesso;
- Il tipo restituito e le specifiche di eccezioni delle funzioni base e derivata deve essere compatibile.

Oltre a questi vincoli, che facevano parte anche del C++98, il C++11 ne prevede un altro.

- I *qualificatori reference* delle funzioni devono essere identici. I qualificatori reference di una funzione membro sono una delle funzionalità meno note del C++11 e dunque non vi sorprendete se non ne avete mai sentito parlare. Rendono possibile limitare l'uso di una funzione membro ai soli valori lvalue oppure rvalue. Le funzioni membro non hanno bisogno di essere virtuali per poterli utilizzare:

```
class Widget {
public:
  …
  void doWork() &;               // questa versione di doWork agisce
                                 // solo quando *this è un lvalue
  void doWork() &&;              // questa versione di doWork agisce
};                               // solo quando *this è un rvalue
…
```

```
Widget makeWidget();            // funzione factory (restituisce
                                un rvalue)

Widget w;                       // normale oggetto (un lvalue)
…
w.doWork();                     // richiama Widget::doWork per
                                lvalue
                                // (per es., Widget::doWork &)

makeWidget().doWork();          // richiama Widget::doWork per
                                rvalue
                                // (per es., Widget::doWork &&)
```

> Parleremo più ampiamente delle funzioni membro con qualificatori reference più avanti; per il momento, notate semplicemente che se una funzione virtuale in una classe base ha un qualificatore reference, gli ovveride di tale funzione nella classe derivata devono avere esattamente lo stesso qualificatore reference. In caso contrario, le funzioni dichiarate esisteranno comunque nella classe derivata, ma non eseguiranno l'override di nulla nella classe base.

Tutti questi requisiti per l'overriding significano che anche piccoli errori possono fare una grande differenza. Il codice contenente errori di overriding può anche essere valido, ma il suo significato non sarà ciò che si poteva immaginare. Pertanto non si può contare sul fatto che i compilatori indicheranno se c'è qualcosa di sbagliato. Per esempio, il codice seguente è perfettamente compilabile e, a prima vista, sembra corretto, ma non contiene override di funzioni virtuali, nemmeno una funzione della classe derivata che si leghi alla funzione della classe base. Riuscite a identificare il problema in ciascun caso, ovvero perché ogni funzione della classe derivata non esegue l'overriding dell'omonima funzione nella classe base?

```
class Base {
public:
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    void mf4() const;
};
```

```
class Derived: public Base {
public:
    virtual void mf1();
    virtual void mf2(unsigned int x);
    virtual void mf3() &&;
    void mf4() const;
};
```

Ecco qualche suggerimento.

- `mf1` è dichiarata `const` in `Base`, ma non in `Derived`.
- `mf2` accetta un `int` in `Base`, mentre un `unsigned int` in `Derived`.
- `mf3` è qualificata lvalue in `Base` ed rvalue in `Derived`.
- `mf4` non è dichiarata `virtual` in `Base`.

Potreste pensare che questo genere di cose vengano rilevate dal compilatore e che dunque non ci sia bisogno di preoccuparsene troppo. A volte questo è vero, ma a volte no. Due dei compilatori in mio possesso hanno accettato questo codice senza battere ciglio, pur con tutti i warning attivati; altri compilatori hanno fornito dei warning relativi ad alcuni di questi problemi, ma non a tutti.

Poiché la dichiarazione di override nella classe derivata è un argomento importante ed è facile sbagliare, il C++11 offre un modo per rendere esplicito il fatto che una funzione in una classe derivata si supponga che debba avere la prevalenza (override) sulla versione presente nella classe base: basta dichiararla `override`. Nel caso specifico di questo esempio, si ottiene la seguente classe derivata:

```
class Derived: public Base {
public:
    virtual void mf1() override;
    virtual void mf2(unsigned int x) override;
    virtual void mf3() && override;
    virtual void mf4() const override;
};
```

Questa classe non passerà la compilazione, naturalmente, poiché, essendo le funzioni scritte in questo modo, i compilatori si lamenteranno dei problemi legati all’overriding.

Questo è esattamente il comportamento desiderato e questo è il motivo per cui si dovrebbero dichiarare con `override` tutte le funzioni che lo richiedono.

Il codice che usa `override` e che invece può essere compilato ha il seguente aspetto (supponendo che l'obiettivo sia che tutte le funzioni di `Derived` eseguano l'overriding di quelle virtuali in `Base`):

```
class Base {
public:
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    virtual void mf4() const;
};

class Derived: public Base {
public:
    virtual void mf1() const override;
    virtual void mf2(int x) override;
    virtual void mf3() & override;
    void mf4() const override;    // l'aggiunta di virtual è OK,
};                                // ma non necessaria
```

Notate che in questo esempio le cose funzionano, almeno in parte, per il fatto che `mf4` viene dichiarata virtuale in `Base`. La maggior parte degli errori legati all'overriding si verifica nelle classi derivate, ma è possibile commettere errori anche nelle classi base.

Una politica che preveda di usare `override` in tutte le classi derivate può fare di più che consentire semplicemente ai compilatori di dire quando l'overriding viene eseguito e quando no. Può anche aiutare a determinare le conseguenze se si prevede di modificare la signature di una funzione virtuale in una classe base. Se le classi derivate usano `override` ovunque, si può semplicemente modificare la signature, ricompilare il sistema, vedere quanto danno si è provocato (per esempio quante classi derivate non vengono compilate) e poi decidere se vale la pena di modificare la signature. Senza `override`, bisognerebbe sperare di avere a disposizione test completi delle unità, poiché, come abbiamo visto, le funzioni virtuali della classe base che falliscono nell'eseguire l'**override** delle funzioni nella classe base, possono non sollevare alcuna lamentela da parte del

compilatore.

Il C++ ha sempre avuto le sue parole chiave, ma il C++11 introduce due *parole chiave contestuali:* `override` e `final`.[2] Queste parole chiave possono essere impiegate solo in determinati contesti. Per esempio, `override` può essere impiegata solo alla fine della dichiarazione di una funzione membro. Ciò significa che se vi è del codice preesistente che già utilizza il nome `override`, non sarà necessario modificarlo per il C++11:

```
class Warning {                    // classe potenzialmente C++98
public:
  …
  void override();                 // lecito in C++98 e C++11
  …                                // (con lo stesso significato)
};
```

Questo è tutto ciò che c'è da dire su `override`, ma c'è ancora qualcosa da dire sui qualificatori reference delle funzioni membro. Avevo promesso altre informazioni specifiche ed eccole.

Se vogliamo scrivere una funzione che accetta solo argomenti lvalue, dichiariamo un parametro reference lvalue `const`:

```
void doSomething(Widget& w);      // accetta solo Widget lvalue
```

Se vogliamo scrivere una funzione che accetta solo argomenti rvalue, dichiariamo un parametro reference rvalue:

```
void doSomething(Widget&& w);      // accetta solo Widget rvalue
```

I qualificatori reference di funzioni membro consentono semplicemente di applicare la stessa distinzione per l'oggetto su cui viene richiamata la funzione membro, per esempio `*this`. È esattamente analogo al `const` alla fine della dichiarazione di una funzione membro, che indica che l'oggetto su cui viene richiamata la funzione membro (per esempio `*this`) è `const`.

La necessità di funzioni membro con qualificatori reference non è comune, ma può sorgere. Per esempio, supponete che la nostra classe `Widget` abbia un dato membro `std::vector` e che vogliamo offrire una funzione di accesso che

conceda ai client un accesso diretto ad esso:

```
class Widget {
public:
  using DataType =
  std::vector<double>;               // vedi Elemento 9
  …                                  // per info su using

  DataType& data() { return
  values; }
  …

private:
  DataType values;
};
```

Questa non è certo la struttura più incapsulata che abbiate mai visto, ma considerate ciò che accade in questo codice client:

```
Widget w;
…

auto vals1 = w.data();             // copia i valori di w in vals1
```

Il tipo restituito da `Widget::data` è un riferimento lvalue (un `std::vector<double>&`, per essere precisi) e poiché i riferimenti lvalue sono definiti come lvalue, stiamo inizializzando `vals1` con un lvalue. `vals1` è pertanto costruita come copia dai valori di `w`, esattamente come dice il commento.

Ora, supponete di avere una funzione per la creazione di `Widget`,

```
Widget makeWidget();
```

e di voler inizializzare una variabile con `std::vector` all'interno del `Widget` restituito da makeWidget:

```
auto vals2 = makeWidget().data(); // copia i valori del
                                  // Widget in vals2
```

Ancora una volta, `Widgets::data` restituisce un riferimento lvalue e, ancora una volta, il riferimento lvalue è un lvalue e dunque, ancora volta, il nostro nuovo oggetto (`vals2`) è costruito per copia utilizzando i valori interni del `Widget`. Questa volta, però, il `Widget` è l'oggetto temporaneo restituito da `makeWidget` (ovvero un rvalue) e, dunque, copiarvi lo `std::vector` è uno spreco di tempo. Sarebbe preferibile spostarlo, ma, poiché `data` restituisce un riferimento lvalue, le regole del C++ richiedono che i compilatori generino codice per una copia (vi è ampio spazio per l'ottimizzazione impiegando la cosiddetta "regola as if ", ma non vi conviene contare sul fatto che un compilatore trovi il modo di sfruttarla).

È necessario un modo per specificare che quando `data` viene richiamato su un `Widget` rvalue, anche il risultato dovrà essere un rvalue. L'utilizzo di qualificatori reference per eseguire l'overloading di `data` per `Widget` lvalue e rvalue lo rende possibile:

```
class Widget {
public:
  using DataType =
  std::vector<double>;
  …

  DataType& data() &                // per Widget lvalue,
  { return values; }                // restituisci lvalue

  DataType data() &&                // per Widget rvalue,
  { return std::move(values); }     // restituisci rvalue
  …

private:
  DataType values;
};
```

Notate i tipi restituiti differenti degli overload di `data`. L'overload lvalue restituisce un riferimento a lvalue (ovvero un lvalue) e l'overload rvalue restituisce un oggetto temporaneo (ovvero un rvalue). Questo significa che il codice client ora si comporta correttamente:

```
auto vals1 = w.data();           // richiama l'overload lvalue per
                                 // Widget::data,
                                 // costruisce per copia vals1

auto vals2 = makeWidget().data(); // richiama l'overload rvalue per
                                  // Widget::data,
                                  // construisce per spostamento
                                  vals2
```

Questo è certamente positivo, ma questo "lieto fine" non vi distragga dal vero argomento di questo Elemento. Ogni volta che si dichiara una funzione in una classe derivata che dovrebbe eseguire l'overriding di una funzione virtuale in una classe base, assicuratevi di dichiarare tale funzione con `override`.

Argomenti da ricordare

- Dichiarare con `override` le funzioni che subiscono overriding.
- I qualificatori di riferimento delle funzioni membro consentono di trattare in modo differente oggetti lvalue e rvalue (`*this`).

Elemento 13 – Preferire i const_iterator agli iterator

I `const_iterator` sono l'equivalente STL dei puntatori a `const`: puntano a valori che non possono essere modificati. La pratica comune di utilizzare `const` ovunque possibile consiglia di utilizzare `const_iterator` ogni volta che è necessario un iteratore e non vi è alcun motivo per dover modificare ciò cui esso punta.

Questo è valido sia in C++98 sia in C++11, ma in C++98, i `const_iterator` avevano solo un supporto parziale. Non era così facile crearli e, una volta creatone uno, era possibile utilizzarlo solo in modo limitato. Per esempio, supponete di voler ricercare in uno `std::vector<int>` la prima occorrenza di 1983 (l'anno in cui il linguaggio ha cambiato nome da "C with Classes" a "C++") e inserire nella stessa posizione il valore 1998 (l'anno in cui è stato

adottato il primo Standard ISO del C++). Se nel vettore non vi è alcun 1983, l'inserimento finirà alla fine del vettore. L'uso degli `iterator` in C++98 era davvero semplice:

```
std::vector<int> values;

…

std::vector<int>::iterator it =
    std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

Ma gli `iterator`, in questo caso, non sono la scelta più adatta, poiché questo codice non modifica mai ciò cui punta un `iterator`. Correggere il codice in modo da utilizzare `const_iterator` dovrebbe essere semplice, ma in C++98 non lo era fatto. Ecco un approccio teoricamente corretto, ma tuttavia errato:

```
typedef std::vector<int>::iterator IterT;            // typedef
std::vector<int>::const_iterator ConstIterT;         // typedef

std::vector<int> values;

…

ConstIterT ci =
  std::find(static_cast<ConstIterT>(values.begin()), // cast
        static_cast<ConstIterT>(values.end()),       // cast
        1983);

values.insert(static_cast<IterT>(ci), 1998);         // può essere
                                                     // incompilabile;
                                                     // vedi sotto
```

I `typedef` non sono obbligatori, naturalmente, ma semplificano la creazione dei

cast nel codice (se vi state chiedendo perché mostro dei `typedef` invece di seguire il consiglio fornito nell'Elemento 9 di utilizzare le dichiarazioni alias, è perché questo esempio mostra codice C++98 e le dichiarazioni alias sono una funzionalità introdotta dal C++11).

Le cast nella chiamata a `std::find` sono presenti poiché `values` è un container non `const` e in C++98 non vi è alcun modo semplice per ottenere un `const_iterator` da un container non `const`. Le cast non sono strettamente necessarie, poiché sarebbe stato possibile ottenere i `const_iterator` in altri modi (per esempio si sarebbe potuto legare `values` a una variabile riferimento-a-**`const`**, poi, nel codice, utilizzare tale variabile al posto di `values`), ma in qualsiasi caso, il processo di ottenere dei `const_iterator` verso gli elementi di un container non `const` prevedeva codice contorto.

Una volta ottenuti i `const_iterator`, le cose peggioravano ulteriormente, poiché in C++98 le posizioni per gli inserimenti (e le cancellazioni) devono essere specificate solo dagli iterator. I `const_iterator` non erano accettabili. Questo è il motivo per cui, nel codice precedente, si trova una cast da un `const_iterator` (che ci siamo preoccupati di ottenere da `std::find`) in un `iterator:`. Il passaggio di un `const_iterator` a `insert` non sarebbe stato accettato dal compilatore.

Per essere onesti, neppure il codice che abbiamo presentato può essere compilato, perché non esiste alcuna conversione portatile da un `const_iterator` a un `iterator`, nemmeno con una `static_cast`. Nemmeno il "martello pneumatico" semantico `reinterpret_cast` sarebbe in grado di risolvere il problema. Non si tratta di una restrizione del C++98; anche in C++11 i `const_iterator`, semplicemente, non possono essere convertiti in `iterator`, indipendentemente dal fatto che questo sembrerebbe solo logico. Vi sono alcuni metodi portatili per generare `iterator` che puntano dove puntano i `const_iterator`, ma non sono né semplici né applicabili universalmente e dunque non è il caso di parlarne. In pratica, spero che il "messaggio" ormai sia chiaro: i `const_iterator` erano talmente un problema in C++98, che non valeva davvero la pena di occuparsene. Alla fine, gli sviluppatori non utilizzavano `const` *ovunque possibile*, ma *ovunque fosse comodo*. E in C++98, i `const_iterator`, semplicemente, non erano comodi.

La situazione è cambiata in C++11. Ora i `const_iterator` sono facili da ottenere e facili da utilizzare. Le funzioni membro container `cbegin` e `cend`

producono `const_iterator` anche per container non `const` e le funzioni membro STL che usano iteratori per identificare le posizioni (per esempio `insert` ed `erase`) in realtà usano `const_iterator`. Rivedere il codice originale C++98 che usa iterator in modo da fargli utilizzare `const_iterator` in C++11 diventa a questo punto molto facile:

```
std::vector<int> values;                    // come prima

…

auto it =                                   // usa cbegin
  std::find(values.cbegin(),values.cend(), // e cend
  1983);

values.insert(it, 1998);
```

Finalmente, ecco del codice che utilizza `const_iterator`, e che è anche comodo!

Vi è un'unica situazione in cui il supporto C++11 di const_iterator dà qualche problema: quando si vuole scrivere codice di libreria il più possibile generico. Tale codice tiene in considerazione che alcuni container e strutture dati di tipo container offrono `begin` e `end` (più `cbegin`, `cend`, `rbegin` e così via) come funzioni *non membro*, invece che come funzioni membro. Questo è il caso degli array standard, per esempio, e anche di alcune librerie esterne con interfacce costituite solo da funzioni libere. Il codice ampiamente generico, pertanto, utilizza funzioni non membro invece di supporre l'esistenza di versioni membro.

Per esempio, si potrebbe generalizzare il codice con cui abbiamo appena lavorato in un template `findAnd Insert` nel seguente modo:

```
template<typename C, typename V>
void findAndInsert(C& container,  // in container, trova
            const V& targetVal,   // la prima occorrenza
            const V& insertVal)   // di targetVal, poi
{                                 // inserisci insertVal
```

```
  using std::cbegin;              // al suo posto
  using std::cend;

  auto it =
  std::find(cbegin(container),    // cbegin non-membro
            cend(container),      // cend non-membro
            targetVal);
  container.insert(it,
  insertVal);
}
```

Questo funziona in C++14, ma, purtroppo, non in C++11. A causa di una svista della standardizzazione, il C++11 ha aggiunto le funzioni non membro `begin` e `end`, ma non le funzioni `cbegin`, `cend`, `rbegin`, `rend`, `crbegin` e `crend`. Il C++14 corregge questa mancanza.

Se, in C++11, intendete scrivere codice davvero generico e nessuna delle librerie che utilizzate fornisce i template mancanti per i non membri `cbegin` e "amici", potete predisporre le vostre implementazioni con una certa facilità. Per esempio, ecco un'implementazione di una `cbegin` non membro:

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
    return std::begin(container); // vedi descrizione, sotto
}
```

Siete sorpresi di vedere che la funzione non membro `cbegin` non richiama la funzione membro `cbegin`? Lo ero anch'io. Ma seguite la logica. Questo template di `cbegin` accetta ogni tipo di argomento che rappresenta una struttura dati di tipo container, `C`, e accede a questo argomento tramite il suo parametro riferimento-a**`-const`,** ovvero `container`. Se `C` è un tipo container convenzionale (per esempio uno `std::vector<int>`), `container` sarà un riferimento alla versione `const` di tale container (ovvero un `const std::vector<int>&`). Richiamando la funzione non membro `begin` (fornita dal C++11) su un container `const`, si ottiene un `const_iterator` e tale iteratore è ciò che viene restituito dal template. Il vantaggio di implementare le cose in questo modo è che funzionano anche per container che offrono una funzione membro `begin` (che, per i

container, è ciò che viene richiamato dalla `begin` non membro del C++11), ma non offrono una funzione membro `cbegin`. Si può pertanto utilizzare questa `cbegin` non membro sui container che supportano direttamente solo `begin`.

Questo template funziona anche se `C` è di tipo array standard. In tal caso, `container` diviene un riferimento a un array `const`. Il C++11 fornisce una versione del `begin` non membro specializzata per gli array, che restituisce un puntatore al primo elemento dell'array. Gli elementi di un array `const` sono `const`, pertanto il puntatore restituito dalla `begin` non membro per un array `const` è un puntatore a `const` e un puntatore a `const` è, in realtà, un `const_iterator` per l'array (per sapere come un template può essere specializzato per gli array standard, consultate la discussione che trovate nell'Elemento 1 sulla deduzione del tipo nei template che prendono parametri riferimento ad array).

Ma torniamo alla "sostanza". Questo Elemento intende incoraggiarvi a utilizzare i `const_iterator` ogni volta che potete. La motivazione fondamentale (usare `const` ogni volta che ha senso) è possibile in C++11, mentre in C++98 semplicemente non era pratica quando si utilizzavano gli operatori. In C++11, è notevolmente comoda e in C++14 ottiene anche alcune aggiunte, che erano state trascurate in C++11.

Argomenti da ricordare

- Preferire i `const_iterator` agli `iterator`.
- Nel codice che deve essere massimamente generico, preferite le versioni non membro di `begin`, `end`, `rbegin` e così via, rispetto alle relative funzioni membro.

Elemento 14 – Dichiarare noexcept le funzioni che non emettono eccezioni

In C++98, le specifiche delle eccezioni erano un po' esigenti. Occorreva riepilogare tutti i tipi di eccezione che una funzione poteva emettere, e così, se cambiava l'implementazione della funzione, poteva dover essere modificata

anche la specifica dell'eccezione. Ma tale modifica poteva avere effetti devastanti sul codice client, poiché i chiamanti potevano dipendere dalla specifica originale dell'eccezione. I compilatori, in genere, non offrivano alcun aiuto nel mantenere la coerenza fra le implementazioni della funzione, le specifiche delle eccezioni e il codice client. La maggior parte dei programmatori, alla fine, aveva deciso che, in C++98, le specifiche delle eccezioni non era proprio il caso di utilizzarle.

Durante il lavoro sul C++11, è emerso un consenso sul fatto che quello che contava davvero in termini di emissione delle eccezioni da parte di una funzione fosse che potesse o non potesse emetterne. Bianco o nero, una funzione poteva emettere un'eccezione oppure doveva garantire di non volerne emettere. Questa dicotomia rappresenta la base delle specifiche delle eccezioni in C++11, che sostituiscono, essenzialmente, quelle del C++98. Le specifiche delle eccezioni in stile C++98 rimangono valide, ma sono sconsigliate. In C++11, il `noexcept` non condizionale è dedicato alle funzioni che garantiscono di non emettere eccezioni.

Il fatto che una funzione debba o meno essere dichiarata `noexcept` è una questione di progettazione dell'interfaccia. Il comportamento di una funzione in termini di emissione di eccezioni è di interesse chiave per i client. I chiamanti possono interrogare lo status `noexcept` di una funzione e i risultati di questa richiesta possono influenzare la sicurezza o l'efficienza dell'eccezione nel codice chiamante. Pertanto, il fatto che una funzione sia `noexcept`, è un'informazione altrettanto importante quanto lo è il fatto che una funzione membro è const. Non dichiarare una funzione come `noexcept` quando si sa che non emetterà un'eccezione significa specificare male l'interfaccia.

Ma vi è un'ulteriore incentivo ad applicare `noexcept` alle funzioni che non producono eccezioni: permette ai compilatori di generare codice oggetto migliore. Per comprenderne il motivo, è utile esaminare la differenza fra i modi C++98 e C++11 per dire che una funzione non emetterà eccezioni. Considerate una funzione `f` che promette ai chiamanti che non emetterà mai un'eccezione. I due modi per esprimerlo sono:

```
int f(int x) throw();      // nessuna eccezione da: stile C++98

int f(int x) noexcept;     // nessuna eccezione da: stile C++11
```

Se, runtime, un’eccezione lascia f, la specifica dell’eccezione di `f` viene violata. Con la specifica dell’eccezione C++98, lo stack di chiamata viene risalito fino al chiamante di `f` e, dopo alcune azioni irrilevanti, l’esecuzione del programma si conclude. Con la specifica delle eccezioni C++11, il comportamento runtime è leggermente differente: lo stack *può essere risalito* prima che l’esecuzione del programma termini.

La differenza sottile tra la *certezza* e la *possibilità* di risalire lo stack ha un impatto notevolissimo sulla generazione del codice. In una funzione `noexcept`, gli ottimizzatori non hanno la necessità di mantenere lo stack di runtime in uno stato risalibile in attesa che un’eccezione si propaghi fuori dalla funzione, né devono garantire che gli oggetti di una funzione `noexcept` vengano distrutti in ordine inverso rispetto alla costruzione qualora un’eccezione dovesse lasciare la funzione. Le funzioni con la specifica per eccezioni “`throw()`” non hanno questa flessibilità di ottimizzazione, così come le funzioni che non hanno alcuna specifica per eccezioni. La situazione può essere riepilogata nel seguente modo:

```
RetType function(params)
noexcept;                          // molto ottimizzabile

RetType function(params) throw(); // meno ottimizzabile

RetType function(params);         // meno ottimizzabile
```

Anche solo questo è un motivo sufficiente per dichiarare le funzioni `noexcept` ogni volta che si sa che non produrranno eccezioni.

Per alcune funzioni, il problema è anche più rilevante. Le operazioni di spostamento sono l’esempio principale. Supponete di avere una base di codice C++98 che fa uso di un `std::vector<Widget>`. I `Widget` vengono aggiunti allo `std::vector` di tanto in tanto tramite `push_back`:

```
std::vector<Widget> vw;

…

Widget w;
```

```
…                                   // lavora su w
vw.push_back(w);                    // aggiunge w a vw
…
```

Supponete che questo codice funzioni e che non abbiate interesse a modificarlo per il C++11. Tuttavia, volete sfruttare il fatto che la semantica di spostamento del C++11 può migliorare le prestazioni del codice preesistente quando sono coinvolti tipi spostabili. Pertanto, vi assicurate che `Widget` contenga le operazioni di spostamento, scrivendole voi stessi oppure facendo in modo che siano esaudite le condizioni per la loro generazione automatica (vedi Elemento 17).

Quando a uno `std::vector` viene aggiunto un nuovo elemento, è possibile che lo `std::vector` non abbia lo spazio per sistemarlo, ovvero che le dimensioni dello `std:: vector` siano uguali alla sua capacità. In questi casi, lo `std::vector` alloca un nuovo frammento di memoria, più grosso, per contenere i suoi elementi e poi trasferisce gli elementi dal frammento di memoria precedente al nuovo. In C++98, il trasferimento viene ottenuto copiando ciascun elemento dalla vecchia area di memoria alla nuova, distruggendo poi gli oggetti nella vecchia area. Questo approccio ha consentito a `push_back` di offrire la garanzia di sicurezza elevata delle eccezioni: se veniva lanciata un'eccezione durante la copia degli elementi, lo stato di `std::vector` rimaneva immutato, poiché nessuno degli elementi nella vecchia area di memoria veniva distrutto finché tutti gli elementi non erano stati copiati con successo nella nuova area di memoria.

In C++11, un'ottimizzazione naturale sarebbe quella di sostituire la copia degli elementi di `std::vector` con degli spostamenti. Sfortunatamente, con questo si corre il rischio di violare la garanzia di sicurezza delle eccezioni di `push_back`. Se dalla vecchia area di memoria vengono trasferiti *n* elementi e poi viene lanciata un'eccezione mentre si sposta l'elemento *n+1*, l'operazione `push_back` non può essere completata. Ma ormai lo `std::vector` originario è stato modificato: *n* dei suoi elementi sono stati trasferiti. Il ripristino dello stato originario potrebbe non essere possibile, poiché anche il tentativo di ri-spostare ciascun oggetto nella memoria originale potrebbe lanciare un'eccezione.

Questo è un problema serio, poiché il comportamento del codice preesistente

potrebbe dipendere dalla garanzia di sicurezza elevata delle eccezioni di std::vector. Pertanto, l'implementazione C++11 non può sostituire "di nascosto" le operazioni di copia di push_back con delle operazioni di spostamento, a meno che si abbia la garanzia che le operazioni di spostamento non lancino eccezioni. In tal caso, sarebbe automaticamente sicuro sostituire le operazioni di copia con quelle di spostamento e l'unico effetto collaterale sarebbe una migliore prestazione.

std::vector::push_back sfrutta questa strategia, "sposta se vuoi, ma copia se devi", e non è l'unica funzione nella Libreria Standard che è in grado di farlo. Altre funzioni che offrono la garanzia di sicurezza elevata delle eccezioni in C++98 (per esempio std::vector::reserve, std::deque::insert e così via) si comportano allo stesso modo. Tutte queste funzioni sostituiscono le chiamate a operazioni di copia del C++98 con altrettante chiamate a operazioni di spostamento in C++11, ma solo se vi è la garanzia che le operazioni di spostamento non producano un'eccezione. Ma come accertare questa garanzia? La risposta, ormai, è ovvia: controllare se l'operazione è dichiarata con noexcept.[3]

Le funzioni swap comprendono un altro caso in cui noexcept è particolarmente utile. swap è un componente chiave di molte implementazioni di algoritmi nella STL ed è comunemente impiegato anche dagli operatori di assegnamento per copia. Il suo ampio uso fa sì che le ottimizzazioni offerte da noexcept siano particolarmente utili. È interessante notare che il motivo per cui gli swap nella Libreria Standard sono noexcept talvolta dipende dal fatto che gli swap definiti dall'utente siano noexcept. Per esempio, le dichiarazioni per gli swap su array e std::pair nella Libreria Standard sono:

```
template <class T, size_t N>
void swap(T (&a)[N],      // vedi
          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // sotto

template <class T1, class T2>
struct pair {
…
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                   noexcept(swap(second, p.second)));
…
};
```

Queste funzioni sono *noexcept condizionali*: il motivo per cui sono noexcept

dipende dal fatto che le espressioni all’interno delle clausole `noexcept` siano `noexcept`. Per esempio, dati due array `Widget`, l’esecuzione dello `swap` su di essi è `noexcept` solo se lo `swap` dei singoli elementi dell’array è `noexcept`, ovvero se lo `swap` per i `Widget` è `noexcept`. Pertanto è l’autore dello `swap` per i `Widget` a determinare se lo `swap` per gli array di `Widget` è `noexcept`. Questo, a sua volta, determina il fatto che altri `swap`, come quello per gli array di array di `Widget` siano `noexcept`. Analogamente, anche il fatto che lo `swap` di due oggetti `std::pair` contenenti `Widget` sia `noexcept` dipende dal fatto che lo `swap` per i `Widget` sia `noexcept`. Il fatto che lo scambio dei dati di alto livello sia `noexcept` solo se lo scambio dei loro costituenti di basso livello è `noexcept` dovrebbe motivarvi a offrire funzioni `swap noexcept` ogni volta che sia possibile.

A questo punto spero che vi sentiate particolarmente interessati alle opportunità di ottimizzazione consentite da `noexcept`. Purtroppo, devo consigliarvi di contenere il vostro entusiasmo. L’ottimizzazione è importante, ma la correttezza è ancora più importante. Ho indicato all’inizio di questo Elemento che `noexcept` fa parte dell’interfaccia di una funzione, pertanto si dovrebbe dichiarare una funzione `noexcept` solo se si vuole aderire a un’implementazione `noexcept` a lungo termine. Se dichiarate una funzione `noexcept` e poi rinnegate questa decisione, le vostre opzioni sono davvero poche. Potreste togliere il `noexcept` dalla dichiarazione della funzione (ovvero modificare la sua interfaccia) correndo così il rischio di non far più funzionare il codice client. Potreste cambiare l’implementazione in modo che un’eccezione possa sfuggire, mantenendo però la specifica d’eccezione originaria (ora errata). In questo caso, il programma verrà chiuso qualora un’eccezione tentasse effettivamente di lasciare la funzione. Oppure potete rassegnarvi alla vostra implementazione esistente, abbandonando del tutto l’intenzione di cambiare implementazione. Nessuna di queste possibilità è particolarmente attraente.

Il problema è che la maggior parte delle funzioni è *neutrale rispetto alle eccezioni*. Tali funzioni non lanciano alcuna eccezione, almeno non direttamente, mentre le funzioni che esse richiamano potrebbero farlo. Quando si verifica questa situazione, una funzione neutrale rispetto alle eccezioni concede all’eccezione emessa il passaggio a un gestore che si trova più in alto nella catena delle chiamate. Le funzioni neutrali rispetto alle eccezioni non sono mai `noexcept`, poiché possono emettere tali eccezioni “di passaggio”. La maggior parte delle funzioni, pertanto, in modo piuttosto appropriato, non ha la designazione `noexcept`.

Alcune funzioni, tuttavia, hanno un'implementazione che, per natura, non emette eccezioni; e per qualche altra funzione, in particolare le operazioni di spostamento e `swap`, il fatto di essere `noexcept` può rappresentare un vantaggio tale che vale la pena di implementarle, quando possibile, in modo `noexcept`.[4] Quando si ha la certezza che una funzione non emetta mai eccezioni, bisognerebbe assolutamente dichiararla `noexcept`.

Notate che ho detto che alcune funzioni hanno implementazioni `noexcept` *naturali*. Snaturare l'implementazione di una funzione in modo da permettere una dichiarazione `noexcept` è come un cane che si morde la coda. È come mettere il carro davanti ai buoi. È come vedere un albero e non l'intera foresta. È come... insomma, scegliete voi la metafora che preferite. Se l'implementazione di una certa funzione può lanciare eccezioni (per esempio richiamando una funzione che potrebbe lanciarne), i salti mortali che dovrete fare per nasconderla ai chiamanti (per esempio individuare tutte le eccezioni e sostituirle con altrettanti codici di stato o specifici valori restituiti dalla funzione) non solo complicheranno l'implementazione della funzione, ma complicheranno anche il codice nei punti di chiamata. Per esempio, i chiamanti possono dover controllare i codici di stato o i particolari valori restituiti dalla funzione. Il costo runtime di queste complicazioni (ulteriori ramificazioni, funzioni più grandi, che poggiano più pesantemente sulla cache delle istruzioni e così via) potrebbero superare di gran lunga qualsiasi accelerazione ottenibile tramite `noexcept`. In più sareste congestionati da codice sorgente sempre più difficile da comprendere e aggiornare. Un perfetto esempio di cattiva ingegnerizzazione del software.

Per alcune funzioni, il fatto di essere `noexcept` è talmente importante che lo sono per default. In C++98, era considerato cattivo stile di programmazione consentire alle funzioni di deallocazione della memoria (come `operator delete` e `operator delete[]`) e ai distruttori di emettere eccezioni, e in C++11 questa regola stilistica è stata promossa a regola del linguaggio. Per default, tutte le funzioni di allocazione della memoria e tutti i distruttori, sia quelli definiti dall'utente sia quelli generati dal compilatore, sono implicitamente `noexcept`. Non vi è alcun bisogno di dichiararli `noexcept` (potete farlo e non succede nulla, semplicemente sembrerà un po' strano). L'unico caso in cui un distruttore non è implicitamente `noexcept` è quando un dato membro di una classe (compresi i membri ereditati e quelli contenuti all'interno di altri dati membri) è di un tipo che stabilisce esplicitamente che il suo distruttore possa emettere eccezioni (per esempio, lo dichiara " `noexcept(false)` "). Tali distruttori sono poco comuni.

Non ve ne è nessuno nella Libreria Standard e se il distruttore per un oggetto utilizzato dalla Libreria Standard (per esempio, perché è in un container o è stato passato a un algoritmo) emette un'eccezione, il comportamento del programma è indefinito.

Vale la pena di notare che alcuni progettisti di interfacce per librerie distinguono fra funzioni con *contratti ampi* e funzioni con *contratti stretti*. Una funzione con un contratto ampio non ha precondizioni. Tale funzione può essere richiamata indipendentemente dallo stato del programma e senza vincoli agli argomenti che le possono passare i chiamanti.[5] Le funzioni con contratto ampio non esibiscono mai comportamenti indefiniti.

Le funzioni che non hanno un contratto ampio, hanno un contratto stretto. Per queste funzioni, se viene violata una precondizione, i risultati sono indefiniti.

Se scrivete una funzione con un contratto ampio e sapete che non emetterà eccezioni, è facile seguire il consiglio di questo Elemento e dichiararla `noexcept`. Per le funzioni con contratti stretti, la situazione è più problematica. Per esempio, immaginate di dover scrivere una funzione `f` che accetta un parametro `std::string` e supponete che l'implementazione naturale di `f` non lanci mai un'eccezione. Ciò suggerisce che `f` debba essere dichiarata `noexcept`.

Ora supponete che `f` abbia una precondizione: la lunghezza del suo parametro std ::string non deve superare i 32 caratteri. Se `f` dovesse essere richiamata con una `std::string` la cui lunghezza è maggiore di 32, il comportamento sarebbe indefinito, poiché *per definizione* la violazione di una precondizione porta a un comportamento indefinito. `f` non ha alcun obbligo di controllare questa precondizione, poiché le funzioni possono sempre presupporre che le proprie precondizioni siano soddisfatte (sono i chiamanti a essere responsabili del fatto che tali assunzioni siano valide). Anche con questa precondizione, quindi, il fatto di dichiarare `f` come `noexcept` sembra appropriato:

```
void f(const std::string& s) noexcept;   // precondizione:
                                         // s.length() <= 32
```

Ma supponete che gli implementatori di `f` scelgano di controllare le violazioni della precondizione. Il controllo non è obbligatorio, ma non è neanche proibito, e poi il controllo della precondizione può anche essere utile, per esempio durante il collaudo del sistema. Il debug di un'eccezione lanciata è generalmente più

facile rispetto a tentare di risalire genericamente alla causa di un comportamento indefinito. Ma come dovrà essere indicata la violazione di una precondizione in modo che un test o un gestore di errori del client possa rilevarla? Un approccio immediato potrebbe essere quello di lanciare un'eccezione "la precondizione è stata violata", ma se `f` è dichiarata `noexcept`, questo sarebbe impossibile: lanciare un'eccezione porterebbe alla chiusura del programma. Per questo motivo, i progettisti di librerie che distinguono fra contratti ampi e stretti generalmente riservano `noexcept` alle funzioni con contratti ampi.

Come punto finale, consentitemi di approfondire la mia osservazione precedente sul fatto che i compilatori, generalmente, non offrono alcun aiuto nell'identificare le incoerenze fra le implementazioni delle funzioni e le specifiche delle loro eccezioni. Considerate il seguente codice, perfettamente legittimo:

```
void setup();                         // funzioni definite altrove
void cleanup();

void doWork() noexcept
{
  setup();                            // predisposizione del lavoro
  …                                   // lavoro
  cleanup();                          // operazioni finali
}
```

Qui, `doWork` è dichiarata `noexcept`, anche se richiama due funzioni non-`noexcept`: `setup` e `cleanup`. Questa sembra una contraddizione, ma potrebbe essere che `setup` e `cleanup` documentino il fatto che non emettono mai eccezioni, anche se non sono dichiarate in questo modo. Ci potrebbe essere un buon motivo per non dichiararle `noexcept`. Per esempio, potrebbero far parte di una libreria scritta in C (anche le funzioni della Libreria Standard C che sono state trasferite nel namespace `std` non hanno le specifiche per le eccezioni, per esempio, `std::strlen` non è dichiarata `noexcept`). Oppure potrebbero far parte di una libreria C++98 che ha deciso di non utilizzare le specifiche per eccezioni C++98 e non è ancora stata rielaborata per C++11.

Poiché esistono motivi legittimi per cui le funzioni `noexcept` contino su codice che non offre la garanzia `noexcept`, il linguaggio C++ permette l'esistenza di

tale codice e i compilatori, generalmente, non emettono warning per queste situazioni.

Argomenti da ricordare

- noexcept fa parte dell'interfaccia di una funzione e questo significa che i chiamanti possono dipendere da esso.
- Le funzioni noexcept sono maggiormente utilizzabili rispetto alle funzioni non-noexcept.
- noexcept è particolarmente utile per le operazioni di spostamento, gli `swap`, le funzioni di deallocazione della memoria e i distruttori.
- La maggior parte delle funzioni non è noexcept; ha un comportamento neutrale rispetto alle eccezioni.

Elemento 15 – Usare `constexpr` quando possibile

Se esistesse un premio per la parola chiave più confusa del C++11, probabilmente verrebbe vinta da `constexpr`. Quando viene applicata agli oggetti, si tratta sostanzialmente di una variante di `const`, ma quando viene applicata alle funzioni, ha un significato piuttosto differente. Vale la pena di togliere di mezzo ogni dubbio, poiché quando `constexpr` corrisponde a ciò che intendete esprimere, sicuramente deciderete di utilizzarla.

Teoricamente, `constexpr` indica un valore che non solo è costante, ma è anche noto durante la compilazione. Questa "teoria" è però solo una parte del ragionamento, poiché quando `constexpr` viene applicata alle funzioni, le cose sono più sfumate di quanto si possa immaginare. Scusatemi se vi rovino il finale a sorpresa, perché devo assolutamente dirvi che non potete presumere che i risultati delle funzioni `constexpr` siano `const`, né potrete dare per scontato che i loro valori siano noti durante la compilazione. Particolarmente interessante è il fatto che queste cose sono semplici *caratteristiche.* È *bene* che le funzioni `constexpr` non abbiano l'obbligo di produrre risultati che sono `const` o noti durante la compilazione!

Ma partiamo dagli oggetti `constexpr`. Tali oggetti sono, in effetti, `const` e, in effetti, contengono valori noti al momento della compilazione. Tecnicamente, i loro valori vengono determinati durante la *traduzione* e la traduzione è costituita non solo dalla compilazione, ma anche dal linking. A meno che per lavoro scriviate compilatori o linker C++, questo non avrà per voi alcuna conseguenza e pertanto potete tranquillamente programmare come se i valori degli oggetti `constexpr` fossero determinati durante la compilazione.

I valori noti durante la compilazione presentano dei vantaggi. Possono essere collocati in memoria di sola lettura, per esempio, e, in particolare per gli sviluppatori di sistemi embedded, questa può essere una caratteristica di notevole importanza. Di applicabilità più ampia è il fatto che i valori interi che sono costanti e noti durante la compilazione, possono essere utilizzati in contesti in cui il C++ richiede un'*espressione costante intera.* Tali contesti comprendono la specifica delle dimensioni di un array, gli argomenti di un template intero (comprese le lunghezze degli oggetti `std::array`), i valori degli enumeratori, gli specificatori di allineamento e molti altri ancora. Se volete utilizzare una variabile per questo genere di cose, certamente vorrete dichiararla `constexpr`, poiché a questo punto i compilatori sapranno che il suo valore è noto in fase di compilazione:

```
int sz;                              // variabile non-constexpr

…

constexpr auto arraySize1 = sz;      // errore! Il valore di sz
                                     // non è noto al momento della
                                     compilazione

std::array<int, sz> data1;           // errore! stesso problema

constexpr auto arraySize2 = 10;      // OK, 10 è una
                                     // costante nota in fase di
                                     compilazione

std::array<int, arraySize2>
data2;                               // OK, arraySize2

                                     // è constexpr
```

Notate che `const` non offre la stessa garanzia di constexpr, poiché gli oggetti `const` non devono essere inizializzati con dei valori durante la compilazione:

```
int sz;                             // come prima

…

const auto arraySize = sz;          // OK, arraySize è
                                    // una copia const di sz

std::array<int, arraySize> data;    // errore! Il valore di arraySize
                                    // non è noto in fase di
                                    compilazione
```

In poche parole, tutti gli oggetti `constexpr` sono `const`, mentre non tutti gli oggetti `const` sono anche `constexpr`. Se volete che i compilatori garantiscano che una variabile abbia un valore che può essere utilizzato in contesti che richiedono vincoli in fase di compilazione, lo strumento da impiegare è `constexpr`, non `const`.

Gli scenari di utilizzo per gli oggetti `constexpr` divengono più interessanti coinvolgendo le funzioni `constexpr`. Tali funzioni producono costanti note in fase di compilazione *quando vengono richiamate con costanti note in fase di compilazione*. Se vengono richiamate con valori non noti, se non runtime, produrranno valori runtime. A questo punto potreste chiedervi a che cosa servono, ma questo non è un buon approccio. Ecco invece come occorre valutarle.

- Le funzioni `constexpr` possono essere utilizzate in contesti che richiedono l'impiego di costanti note in fase di compilazione. Se i valori degli argomenti passati a una funzione `constexpr` in questo contesto sono noti durante la compilazione, il risultato verrà precalcolato durante la compilazione. Se il valore di qualcuno degli argomenti non è noto durante la compilazione, il codice verrà rifiutato.
- Quando una funzione `constexpr` viene richiamata con uno o più valori che non sono noti durante la compilazione, si comporta come una normale funzione, calcolando i suoi risultati runtime. Questo significa che non

avete bisogno di impiegare due funzioni per svolgere la stessa operazione, una per le costanti note in fase di compilazione e una per tutti gli altri valori. Si occupa di tutto la funzione `constexpr`.

Supponete di aver bisogno di una struttura dati che contenga i risultati di un esperimento che può essere eseguito in svariati modi. Per esempio, nel corso di un esperimento il livello di illuminazione può essere elevato, basso o spento (ma potrebbe trattarsi della velocità di un ventilatore, della temperatura e così via). Se vi sono *n* condizioni ambientali rilevanti per l'esperimento, ognuna delle quali prevede tre possibili stati, il numero di combinazioni è pari a 3^n. La memorizzazione dei risultati sperimentali di tutte le combinazioni di condizioni richiederebbe pertanto una struttura di dati con spazio sufficiente per 3^n valori. Supponendo che ciascun risultato sia un `int` e che sia noto (o che possa essere calcolato) durante la compilazione, una scelta ragionevole sarebbe quella di impiegare la struttura dati `std::array`. Ma avremo bisogno di un modo per calcolare 3^n durante la compilazione. La Libreria Standard C++ fornisce `std::pow`, che è la funzionalità matematica di cui abbiamo bisogno, ma, per i nostri scopi, questa presenta due problemi. Innanzitutto, `std::pow` funziona su tipi in virgola mobile e abbiamo bisogno di un risultato intero. In secondo luogo, `std::pow` non è `constexpr` (per esempio non vi è alcuna garanzia che restituisca effettivamente un risultato al momento della compilazione anche se viene richiamata con valori noti al momento della compilazione), pertanto non possiamo utilizzarla per specificare la dimensione dello `std::array`.

Fortunatamente, possiamo scrivere la `pow` di cui abbiamo bisogno. La vedremo fra poco, ma innanzitutto vediamo come potrebbe essere dichiarata e utilizzata:

```
constexpr                                  // pow è una funzione constexpr
int pow(int base, int exp) noexcept        // che non lancia eccezioni
{
  …                                        // implementazione di seguito
}

constexpr auto numConds = 5;               // numero condizioni

std::array<int, pow(3, numConds)> results; // results ha
```

```
                                            // 3^numConds
                                            // elementi
```

Ricordiamo che il `constexpr` davanti a `pow` non dice che `pow` restituisce un valore `const`; dice solo che se `base` ed `exp` sono costanti note al momento della compilazione, il risultato di `pow` può essere utilizzato come una costante di compilazione. Se `base` e/o `exp` non sono costanti note al momento della compilazione, il risultato di `pow` verrà calcolato runtime. Ciò significa che `pow` può essere richiamata non solo per calcolare in fase di compilazione le dimensioni di uno `std::array`, ma può essere anche richiamata in contesti runtime come il seguente:

```
auto base = readFromDB("base");   // leggi questi valori
auto exp =
readFromDB("exponent");           // runtime

auto baseToExp = pow(base, exp);  // richiama la funzione pow
                                  // runtime
```

Poiché le funzioni constexpr devono essere in grado di restituire risultati in fase di compilazione quando vengono richiamate con valori noti in fase di compilazione, vengono imposte delle restrizioni sulla loro implementazione. Le restrizioni differiscono fra C++11 e C++14.

In C++11, le funzioni `constexpr` possono contenere un'unica istruzione eseguibile: un `return`. Questa limitazione può sembrare più limitante di quanto non sia, poiché possono essere impiegati due trucchi, per estendere l'espressività delle funzioni `constexpr` oltre quanto si potrebbe inizialmente immaginare. Innanzitutto, si può utilizzare l'operatore condizionale "`?`" e al posto delle istruzioni `if-else`; in secondo luogo, si può utilizzare la ricorsione al posto dei cicli. Pertanto, `pow` può essere implementata nel seguente modo:

```
constexpr int pow(int base, int exp) noexcept
{
return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

Questo funziona, ma difficilmente questa soluzione può sembrare interessante per un normale programmatore di funzioni. In C++14, le restrizioni a cui devono

sottostare le funzioni `constexpr` hanno pretese molto inferiori e, pertanto, diviene possibile la seguente implementazione:

```
constexpr int pow(int base, int exp) noexcept // C++14
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;


  return result;
}
```

Le funzioni `constexpr` possono solo prendere e restituire *tipi letterali*, il che significa sostanzialmente tipi che abbiano un valore determinato durante la compilazione. In C++11, possono essere impiegati tutti i tipi standard, tranne `void`, ma anche i tipi definiti dall'utente possono essere letterali, poiché anche i costruttori e le altre funzioni membro possono essere `constexpr`:

```
class Point {
public:
  constexpr Point(double xVal = 0, double yVal = 0) noexcept
  : x(xVal), y(yVal)
  {}


  constexpr double xValue() const noexcept { return x; }
  constexpr double yValue() const noexcept { return y; }


  void setX(double newX) noexcept { x = newX; }
  void setY(double newY) noexcept { y = newY; }
private:
  double x, y;
};
```

Qui, il costruttore di `Point` può essere dichiarato `constexpr`, poiché se gli argomenti che gli vengono passati sono noti durante la compilazione, anche il valore dei dati membro del `Point` costruito può essere noto durante la compilazione. I `Point` così inizializzati possono pertanto essere `constexpr`:

```
constexpr Point p1(9.4, 27.7);    // OK, "usa" il costruttore
                                  // constexpr durante la
```

```
                                  compilazione

constexpr Point p2(28.8, 5.3);    // funziona
```

Analogamente, anche i getter `xValue` e `yValue` possono essere `constexpr`, poiché se vengono richiamati su un oggetto `Point` con un valore noto durante la compilazione (per esempio un oggetto `Point` che è `constexpr`), i valori dei dati membro `x` e `y` possono essere noti durante la compilazione. Ciò consente di scrivere funzioni `constexpr` che richiamano i getter di `Point` e di inizializzare gli oggetti `constexpr` con i risultati di queste funzioni:

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() +
  p2.xValue()) / 2,                // richiama funzioni
    (p1.yValue() + p2.yValue()) /
    2 };                           // membro constexpr
}

constexpr auto mid = midpoint(p1,
p2);                               // inizializza l'oggetto
                                   // constexpr col risultato della
                                   // funzione constexpr
```

Tutto questo è molto interessante. Significa che l'oggetto `mid`, anche se la sua inizializzazione richiede chiamate a costruttori, a getter e a una funzione non membro, può essere creato nella memoria di sola lettura! Significa la possibilità di utilizzare un'espressione come `mid.xValue() * 10` in un argomento di un template o in un'espressione che specifica il valore di un enumeratore![6] Significa che la linea tradizionalmente rigida che separa il lavoro svolto durante la compilazione e quello svolto runtime inizia a farsi meno netta e alcuni calcoli tradizionalmente eseguiti runtime possono migrare alla fase di compilazione. Più codice svolge questa migrazione, maggiore sarà la velocità del software. Sarà la compilazione, però, a richiedere più tempo.

In C++11, due restrizioni impediscono alle funzioni membro di `Point`, `setX` e `setY`, di essere dichiarate `constexpr`. Innanzitutto, modificano l'oggetto su cui

operano e, in C++11, le funzioni membro `constexpr` sono implicitamente `const`. In secondo luogo, restituiscono un valore di tipo `void`, e `void` non è un tipo letterale in C++11. In C++14 entrambe queste restrizioni spariscono e dunque anche i setter di `Point` possono essere constexpr:

```
class Point {
public:
  …

  constexpr void setX(double newX) noexcept          // C++14
  { x = newX; }

  constexpr void setY(double newY) noexcept          // C++14
  { y = newY; }
  …
};
```

Ciò consente di scrivere funzioni come la seguente:

```
// restituisce la riflessione di p rispetto all'origine (C++14)
constexpr Point reflection(const Point& p) noexcept
{
  Point result;                     // crea un Point non-const

  result.setX(-p.xValue());         // imposta i suoi valori x e y
  result.setY(-p.yValue());

  return result;                    // ne restituisce una copia
}
```

Il codice client potrebbe avere il seguente aspetto:

```
constexpr Point p1(9.4, 27.7);    // come sopra
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1,
```

```
p2);

constexpr auto reflectedMid =     // il valore di reflectedMid è
reflection(mid);                  // (-19.1 -16.5) ed è noto
                                  // durante la compilazione
```

Il consiglio di questo Elemento è quello di utilizzare funzioni constexpr ogni volta che sia possibile e, a questo punto, i motivi dovrebbero essere chiari: sia gli oggetti constexpr sia le funzioni constexpr possono essere impiegati in una gamma di contesti più ampia rispetto agli oggetti e alle funzioni che non sono constexpr. Utilizzando constexpr quando possibile, si allarga il ventaglio di situazioni in cui possono essere utilizzati gli oggetti e le funzioni.

È importante notare che constexpr fa parte dell'interfaccia di un oggetto o di una funzione. In pratica, constexpr dice: "Posso essere utilizzato in un contesto in cui il C++ richiede un'espressione costante". Se dichiarate un oggetto o una funzione come constexpr, i client potranno utilizzarlo in tali contesti. Se successivamente decidete che la scelta di utilizzare constexpr è stata un errore e lo si toglie, questo potrebbe far sì che quantità arbitrariamente estese di codice client smettano di essere compilabili (il semplice atto di aggiungere delle operazioni di I/O a una funzione per motivi di debug o di ottimizzazione delle prestazioni può portare a un problema di questo tipo, poiché le istruzioni di I/O generalmente non sono consentite nelle funzioni constexpr). Parte di quel "ogni volta che sia possibile" nel consiglio di "utilizzare funzioni constexpr ogni volta che sia possibile" sta nella vostra intenzione di aderire a lungo termine ai vincoli che ciò impone agli oggetti e alle funzioni cui viene applicato.

Argomenti da ricordare

- Gli oggetti `constexpr` sono `const` e vengono inizializzati con valori noti in fase di compilazione.
- Le funzioni constexpr possono produrre risultati noti in fase di compilazione quando vengono richiamate con argomenti noti in fase di compilazione.
- Gli oggetti e le funzioni constexpr possono essere utilizzati in una gamma di contesti più ampia rispetto agli oggetti e alle funzioni non constexpr.

- constexpr è parte dell’interfaccia di un oggetto o di una funzione.

Elemento 16 – Rendere sicure per i thread le funzioni membro const

Se lavorassimo in un ambito matematico, potremmo voler rappresentare i polinomi tramite una classe. All’interno di questa classe, sarebbe probabilmente utile avere una funzione che calcoli le radici di un polinomio, ovvero i valori per i quali il polinomio vale zero. Tale funzione non modificherebbe il polinomio, pertanto sarebbe naturale dichiararla const:

```
class Polynomial {
public:
  using RootsType =              // struttura dati contenente i valori
    std::vector<double>;         // per i quali il polinomio vale zero
  …                              // (vedi Elemento 9 per info su "using")

  RootsType roots() const;

   …
};
```

Il calcolo delle radici di un polinomio può essere dispendioso e così intendiamo eseguire questo calcolo solo se siamo costretti. E se proprio dobbiamo calcolarlo, certamente non vogliamo ripetere il calcolo più di una volta. Pertanto salveremo in cache le radici del polinomio nel caso dovessimo calcolarle e implementeremo `roots` in modo che restituisca il valore già memorizzato nella cache. Ecco l’approccio base:

```
class Polynomial {

public:
  using RootsType = std::vector<double>;
```

```
RootsType roots() const
{
  if (!rootsAreValid) {           // se la cache con è valida

  …                               // calcola le radici,
                                  // e memorizzale in rootVals
  rootsAreValid = true;
}

  return rootVals;
}

private:
  mutable bool rootsAreValid{     // vedi Elemento 7 per info
  false };
  mutable RootsType rootVals{};   // sugli inizializzatori
};
```

Teoricamente, `roots` non cambia l'oggetto `Polynomial` su cui opera, ma, nell'ambito della gestione della cache, può dover modificare `rootVals` e `rootsAreValid`. Questo è un classico caso di utilizzo di `mutable` e questo è il motivo per cui tale modificatore fa parte delle dichiarazioni di questi dati membro.

Immaginate ora che due thread richiamino simultaneamente `roots` su un oggetto `Polynomial`:

```
Polynomial p;
…
/*----- Thread 1 -----*/          /*------- Thread 2 -------*/

auto rootsOfP = p.roots();        auto valsGivingZero = p.roots();
```

Questo codice client è perfettamente ragionevole. `roots` è una funzione membro `const` e questo significa che rappresenta un'operazione di lettura. Non vi dovrebbe essere alcun problema se più thread svolgono un'operazione di lettura

senza sincronizzarsi. Almeno così dovrebbe essere. In questo caso, però, non è così, poiché all'interno di `roots` uno o entrambi questi thread potrebbero tentare di modificare i dati membro `rootsAreValid` e `rootVals`. Ciò significa che questo codice potrebbe avere thread differenti che leggono e scrivono sulla stessa area di memoria senza sincronizzazione e dunque vi sorgerà un conflitto nell'accesso ai dati. Il risultato è che questo codice ha un comportamento indefinito.

Il problema è che `roots` è dichiarato `const`, ma non gestisce i thread in modo sicuro. La dichiarazione `const` è corretta in C++11 come lo sarebbe in C++98 (il calcolo delle radici di un polinomio non cambia il valore del polinomio), dunque ciò che deve essere corretto è la mancanza di sicurezza nella gestione dei thread.

Il modo più semplice per risolvere il problema è quello consueto: impiegare un `mutex`:

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
{
    std::lock_guard<std::mutex> g(m);    // bloccaggio del mutex

    if (!rootsAreValid)          { // se la cache non è valida

    …                              // calcola/memorizza le radici

      rootsAreValid = true;
  }
  return rootVals;
}                                  // sbloccaggio del mutex

private:
  mutable std::mutex m;
  mutable bool rootsAreValid{ false };
```

```
  mutable RootsType rootVals{};
};
```

Lo `std::mutex m` è dichiarato `mutable`, poiché viene bloccato e sbloccato da funzioni membro non `const`, e all'interno di `roots` (una funzione membro `const`), `m` sarebbe altrimenti considerato un oggetto `const`.

Vale la pena di notare che, poiché `std::mutex` è un tipo *move-only* (ovvero un tipo che può essere spostato ma non copiato), un effetto collaterale dell'aggiunta di `m` a `Polynomial` è il fatto che `Polynomial` perde la possibilità di essere copiato. Comunque può essere spostato.

In alcune situazioni, un mutex è fin esagerato. Per esempio, se non dovete fare altro che contare quante volte viene richiamata una funzione membro, un contatore `std:: atomic` (in cui gli altri thread hanno la garanzia che le operazioni che si verifichino in modo indivisibile, vedi Elemento 40) sarà certamente un approccio meno "costoso" (il fatto che sia effettivamente meno costoso dipende dall'hardware impiegato e dall'implementazione dei mutex nella Libreria Standard). Ecco come potete impiegare uno `std::atomic` per contare le chiamate:

```
class Point {                                    // punto 2D
public:
…

  double distanceFromOrigin() const noexcept     // vedi Elemento 14
  {                                              // per noexcept

    ++callCount;                                 // incremento atomico

    return std::sqrt((x x) + (y y));
  }

private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
```

```
};
```

Come gli `std::mutex`, anche gli `std::atomic` sono tipi move-only, pertanto la presenza di `callCount` in `Point` significa che anche `Point` è move-only.

Poiché spesso le operazioni su variabili `std::atomic` sono meno costose rispetto all'acquisizione/rilascio di un mutex, potreste essere tentati di contare su `std::atomic` più di quanto dovreste. Per esempio, in una classe che salva in cache un `int` il cui calcolo richiede molto tempo, potreste tentare di utilizzare una coppia di variabili `std::atomic` invece di un mutex:

```
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;   // ahi ahi, parte 1
      cacheValid = true;   // ahi ahi, parte 2
      return cachedValue;
  }

}

private:
  mutable std::atomic<bool> cacheValid{ false };
  mutable std::atomic<int> cachedValue;

};
```

Funzionerebbe, ma talvolta le cose sono più complesse di quanto dovrebbero. Considerate i due fatti seguenti.

- Un thread richiama `Widget::magicValue`, vede che `cacheValid` è `false`, esegue due calcoli "costosi" e assegna la loro somma a `cachedValue`.
- A questo punto, un secondo thread richiama `Widget::magicValue`, anch'esso trova `cacheValid` come `false` e quindi svolge gli stessi calcoli "costosi" che il primo thread ha appena terminato (abbiamo parlato di un "secondo thread", ma, in realtà, potrebbe trattarsi di *parecchi* thread).

Tale comportamento è contrario all'obiettivo stesso della cache. Invertendo l'ordine degli assegnamenti a `cachedValue` e `cacheValid` si elimina questo problema, ma il risultato è perfino peggiore:

```
class Widget {
public:
  …


  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cacheValid = true;   // ahi ahi, parte 1
      return cachedValue = val1 + val2;   // ahi ahi, parte 2
    }
  }

  …

};
```

Immaginate che `cacheValid` sia `false`. Ci troviamo nella seguente situazione.

- Un thread richiama `Widget::magicValue` e continua l'esecuzione fino al punto in cui `cacheValid` viene impostato a `true`.
- A questo punto, un secondo thread richiama `Widget::magicValue` e controlla `cacheValid`. Vedendo che è `true`, il thread restituisce il valore di `cachedValue`, anche se il primo thread non ha ancora fatto a tempo ad assegnargli un valore. Il valore restituito, pertanto, è errato.

Questo insegna una lezione. Per un'unica variabile o area di memoria che richieda sincronizzazione, l'uso di un `std::atomic` è inadeguato, ma una volta che si arriva a due o più variabili o aree di memoria che devono essere manipolate come un'unica unità, dovreste impiegare un mutex. Per `Widget::magicValue`, ciò avrebbe il seguente aspetto:

```
class Widget {
public:
```

```
  …

  int magicValue() const
  {
    std::lock_guard<std::mutex> guard(m);     // blocca m

    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;
      cacheValid = true;
      return cachedValue;
    }
  }                                           // sblocca m
  …

private:
  mutable std::mutex m;
  mutable int cachedValue;                    // non più atomica
  mutable bool cacheValid{ false };           // non più atomica
};
```

Ora, questo Elemento si basa sull'assunto che più thread possano eseguire simultaneamente una funzione membro `const` su un oggetto. Se state scrivendo una funzione membro `const` senza un reale motivo (perché potete *garantire* che non vi sarà mai più di un thread a eseguire tale funzione membro su un oggetto) la sicurezza della funzione nei confronti dei thread sarà irrilevante. Per esempio, non è importante che le funzioni membro delle classi progettate per essere esclusivamente mono-thread siano sicure nei confronti dei thread. In tali casi,

potete evitare i costi associati ai mutex e a `std::atomic` così come l'effetto collaterale del rendering delle classi che le contengono. Tuttavia, tali scenari senza thread sono sempre meno comuni e probabilmente in futuro saranno ancora più rari. Si può scommettere che le funzioni membro `const` saranno sempre più soggette a un'esecuzione concorrente e questo è il motivo per cui dovreste garantire che le funzioni membro `const` siano sicure nei confronti dei thread.

Argomenti da ricordare

- Rendete le funzioni membro `const` sicure nei confronti dei thread, a meno che non siate *certi* che non verranno mai utilizzate in un contesto concorrente.
- Le variabili `std::atomic` possono offrire prestazioni migliori rispetto a un mutex, ma sono adatte alla manipolazione di una sola variabile o area di memoria.

Elemento 17 – La generazione di funzioni membro speciali

Nel gergo ufficiale C++, le *funzioni membro speciali* sono quelle che il linguaggio C++ intende generare da solo. Il C++98 ne prevede quattro: il costruttore standard, il distruttore, il costruttore per copia e l'operatore di assegnamento per copia. Qualche dettaglio: queste funzioni vengono generate solo quando sono necessarie, ovvero se vi è del codice che le utilizza senza che esse siano state dichiarate espressamente nella classe. Un costruttore standard viene generato solo se la classe non dichiara alcun costruttore (questo evita ai compilatori di creare un costruttore standard per una classe in cui avete specificato che gli argomenti del costruttore sono obbligatori). Le funzioni membro speciali generate sono implicitamente pubbliche e `inline` e sono anche non-virtuali, a meno che la funzione in questione sia un distruttore in una classe derivata che eredita da una classe base tramite un distruttore virtuale. In tal caso, è virtuale anche il distruttore generato dal compilatore per la classe derivata.

Ma tutto questo lo sapete già. Sono precisazioni che suonano di antico, di

Mesopotamia, di dinastia Shang, di FORTRAN e C++98. I tempi sono cambiati e sono cambiate anche le regole per la generazione delle funzioni membro speciali in C++. È importante tenere in considerazione queste nuove regole, poiché poche cose sono così importanti per un'efficace programmazione in C++ come il fatto di conoscere quando i compilatori inseriscono silenziosamente delle funzioni membro nelle classi.

In C++11, il "club" delle funzioni membro speciali ha due nuovi "iscritti": il costruttore per spostamento e l'operatore di assegnamento per spostamento:

```
class Widget {
public:
  …
  Widget(Widget&& rhs);                 // costruttore per spostamento

  Widget& operator=(Widget&& rhs);      // operatore di assegnamento per spostamento
  …
};
```

Le regole che governano la loro generazione e il loro comportamento sono analoghe a quelle dei loro "parenti" per copia. Le operazioni di spostamento vengono generate solo quando sono necessarie; e quando vengono generate, eseguono spostamenti "membro a membro" per i dati membro non statici della classe. Ciò significa che il costruttore per spostamento costruisce ogni dato membro non statico della classe a partire dal membro corrispondente al suo parametro `rhs` e l'operatore di assegnamento per spostamento assegna ciascun dato membro non statico a partire dal suo parametro. Il costruttore per spostamento costruisce anche le parti della classe base (se ve ne sono) e l'operatore di assegnamento per spostamento assegna anche le parti della classe base.

Ora, quando facciamo riferimento a un'operazione di spostamento che costruisce o assegna un dato membro o una classe base, non vi è alcuna garanzia che tale spostamento si svolga effettivamente. Gli spostamenti "membro a membro" sono, in realtà, più che altro *richieste* di spostamento membro a membro, poiché i tipi per i quali *non è consentito lo spostamento* (ovvero quelli che non offrono un particolare supporto per le operazioni di spostamento, come la maggior parte

delle classi derivate dal C++98) verranno “spostati” tramite operazioni di copia. Il cuore di ogni “spostamento” membro a membro è l’applicazione di `std::move` all’oggetto da cui eseguire lo spostamento e il risultato viene utilizzato durante la risoluzione dell’overload della funzione per determinare se debba essere eseguito uno spostamento o una copia. L’Elemento 23 descrive in dettaglio questo processo. A questo punto, basta ricordare che uno spostamento membro a membro è costituito da operazioni di spostamento per i dati membro e le classi base che supportano un’operazione di spostamento, ma da un’operazione di copia per quelli che non supportano tali operazioni.

Come nel caso delle operazioni di copia, le operazioni di spostamento non vengono generate se provvedete voi stessi a dichiararle. Tuttavia, le esatte condizioni in cui vengono generate differiscono un po’ da quelle per le operazioni di copia.

Le due operazioni di copia sono indipendenti: la dichiarazione dell’una non impedisce ai compilatori di generare l’altra. Pertanto, se dichiarate un costruttore per copia, ma non un operatore di assegnamento per copia, poi scrivete del codice che richiede l’assegnamento per copia, i compilatori genereranno per voi l’operatore di assegnamento per copia. Analogamente, se dichiarate un operatore di assegnamento per copia, ma non un costruttore per copia, e il codice richiede la costruzione di una copia, i compilatori genereranno automaticamente il costruttore per copia. Questo era vero in C++98 e rimane vero in C++11.

Le due operazioni di spostamento, invece, *non* sono indipendenti. Se dichiarate una, ciò dice al compilatore di generare l’altra. Il motivo è che se dichiarate, per esempio, un costruttore per spostamento per la classe, indicate che vi è qualche motivo che spinge a realizzare il costruttore per spostamento, il quale questo dovrebbe essere implementato in modo differente rispetto al semplice spostamento membro a membro che avrebbe generato il compilatore. E se qualcosa non va nella costruzione per spostamento membro a membro, probabilmente ci sarà qualcosa che non va anche nell’assegnamento per spostamento membro a membro. Pertanto, la dichiarazione di un costruttore per spostamento impedisce la generazione di un operatore di assegnamento per spostamento e la dichiarazione di un operatore di assegnamento per spostamento impedisce al compilatore di generare un costruttore per spostamento.

Inoltre, le operazioni di spostamento non vengono generate per le classi che dichiarano esplicitamente un’operazione di copia. Il motivo è che la

dichiarazione di un’operazione di copia (costruzione o assegnamento) indica che il normale approccio alla copia di un oggetto (copia membro a membro) non è appropriata per la classe; pertanto il compilatore considera che se la copia membro a membro non è appropriata per le operazioni di copia, lo spostamento membro a membro probabilmente non sarà appropriato per le operazioni di spostamento.

La cosa vale anche nella direzione opposta. La dichiarazione di un’operazione di spostamento (costruzione o assegnamento) in una classe fa sì che i compilatori disabilitino le operazioni di copia (le operazioni di copia vengono disabilitate cancellandole, vedi Elemento 11). Dopo tutto, se lo spostamento membro a membro non è il modo corretto per spostare un oggetto, non vi è alcun motivo per aspettarsi che la copia membro a membro sia il modo corretto per copiarlo. Può sembrare che ciò possa infrangere il codice C++98, poiché le condizioni sotto le quali vengono consentite le operazioni di copia sono più vincolanti in C++11 rispetto al C++98, ma le cose non stanno così. Il codice C++98 non può contenere operazioni di spostamento, poiché in C++98 non esisteva alcuna operazione di “spostamento” degli oggetti. L’unico modo in cui una classe preesistente poteva prevedere operazioni di spostamento dichiarate dall’utente era che queste fossero state aggiunte per il C++11; in più, le classi modificate per sfruttare la semantica di spostamento devono seguire le regole del C++11 per la generazione delle funzioni membro speciali.

Forse avete sentito parlare della cosiddetta *Regola dei tre.* Tale regola stabilisce che se si dichiara un costruttore per copia o un operatore di assegnamento per copia oppure un distruttore, occorre dichiararli tutti e tre. La regola deriva dall’osservazione che la necessità di modificare il significato di un’operazione di copia deriva quasi sempre dal fatto che la classe esegue un qualche tipo di gestione delle risorse; ciò, quasi sempre, implica il fatto che (1) qualsiasi gestione delle risorse venga eseguita da un’operazione di copia, probabilmente debba essere eseguita anche nell’altra operazione di copia e (2) il distruttore della classe deve partecipare alla gestione della risorsa (normalmente rilasciandola). Tipicamente, la risorsa da gestire è la memoria e questo è il motivo per cui tutte le classi della Libreria Standard che gestiscono la memoria (per esempio i container STL che si occupano della gestione della memoria dinamica) le dichiarano sempre tutte e tre: entrambe le operazioni di copia e il distruttore.

Una conseguenza della Regola dei tre è che la presenza di un distruttore

dichiarato dall’utente indica che la semplice copia membro a membro difficilmente sarà appropriata per le operazioni di copia della classe. Ciò, a sua volta, suggerisce che se in una classe si dichiara un distruttore, le operazioni di copia probabilmente non dovranno essere generate in modo automatico, poiché non svolgerebbero l’operazione corretta. Nel momento in cui è stato adottato il C++98, il significato di questo ragionamento non è stato apprezzato appieno, pertanto in C++98 l’esistenza di un distruttore dichiarato dall’utente non aveva alcun impatto sull’intenzione dei compilatori di generare delle operazioni di copia. La stessa cosa vale in C++11, ma solo perché la restrizione delle condizioni sotto le quali vengono generate le operazioni di copia avrebbe reso inutilizzabile troppo codice preesistente.

Il ragionamento su cui si basa la Regola dei tre rimane valido; tuttavia, questo, combinato con l’osservazione che la dichiarazione di un’operazione di copia preclude la generazione implicita delle operazioni di spostamento, motiva il fatto che il C++11 *non* genera le operazioni di spostamento per una classe ove vi sia un distruttore dichiarato dall’utente.

Pertanto, le operazioni di spostamento vengono generate per le classi (quando necessario) solo se valgono i tre fatti seguenti:

- nella classe non è dichiarata alcuna operazione di copia;
- nella classe non è dichiarata alcuna operazione di spostamento;
- nella classe non è dichiarato alcun distruttore.

Regole analoghe possono essere estese (più o meno) alle operazioni di copia, poiché il C++11 sconsiglia la generazione automatica delle operazioni di copia per le classi che dichiarano le operazioni di copia o un distruttore. Questo significa che se avete del codice che dipende dalla generazione delle operazioni di copia nelle classi che dichiarano un distruttore o una delle operazioni di copia, dovreste considerare l’aggiornamento di tali classi per eliminare la dipendenza. Ammettendo che il comportamento delle funzioni generate dal compilatore sia corretto (ovvero se la copia membro a membro dei dati membro non statici della classe è proprio ciò che volete), il compito è semplice, poiché la parte “= `default`” del C++11 consente di chiederlo esplicitamente:

```
class Widget {
public:
```

```
  …
  ~Widget();                        // distr. dichiarato dall'utente

  …                                 // costr. per copia di default
  Widget(const Widget&) =           // OK
  default;

  Widget&                           // ass. per copia di default
    operator=(const Widget&) =      // behavior è OK
    default;
  …
};
```

Questo approccio è utile, spesso, nelle classi basi polimorfiche, ovvero quelle classi che definiscono interfacce attraverso le quali vengono manipolati gli oggetti delle classi derivate. Le classi base polimorfiche, normalmente, hanno dei distruttori virtuali, poiché se non li avessero, alcune operazioni (per esempio l'uso di `delete` o `typeid` su un oggetto di una classe derivata attraverso un puntatore o un riferimento alla classe base) produrrebbe risultati indefiniti o fuorvianti. A meno che una classe erediti un distruttore che sia già virtuale, l'unico modo per rendere virtuale un distruttore consiste nel dichiararlo esplicitamente in questo modo. Spesso, l'implementazione standard sarà corretta e "= default" è un buon modo per esprimerlo. Tuttavia, un distruttore dichiarato dall'utente sopprime la generazione delle operazioni di spostamento e dunque quando deve essere supportata la spostabilità, "= default" spesso trova una seconda applicazione. La dichiarazione delle operazioni di spostamento disattiva le operazioni di copia, e dunque se si cerca anche la possibilità di eseguire copie, un ulteriore "= default" svolge il compito:

```
class Base {
public:
  virtual ~Base() = default;        // rende virtuale il distr.
  Base(Base&&) = default;           // supporto dello spostamento
  Base& operator=(Base&&) =
  default;

  Base(const Base&) = default;      // supporto della copia
```

```
  Base& operator=(const Base&) =
  default;
   …
};
```

In realtà, anche se avete una classe per la quale i compilatori vogliono generare le operazioni di copia e di spostamento e nella quale le funzioni generate si comportano nel modo desiderato, potete scegliere di adottare una politica che consiste nel dichiararle comunque e utilizzare “= default” per le loro definizioni. Richiede un po’ più lavoro, ma rende più chiare le intenzioni e può aiutare a eliminare alcuni subdoli bug. Supponete di avere una classe che rappresenta una tabella di stringhe, per esempio una struttura dati che permetta una ricerca rapida di valori stringa tramite un ID intero:

```
class StringTable {
public:
  StringTable() {}
…                           // funzioni per inserimento,
                            cancellazione, ricerca,
                            // ecc., ma nessuna funzionalità di
                            copia/spostamento/distr.

private:
  std::map<int,
  std::string> values;
};
```

Se supponiamo che la classe non dichiari alcuna operazione di copia, alcuna operazione di spostamento e alcun distruttore, i compilatori genereranno automaticamente queste funzioni, qualora vengano impiegate. Questo è molto comodo.

Ma supponete che, qualche tempo dopo, venga deciso che sarebbe utile registrare il momento in cui le funzioni di default eseguono la costruzione e la distruzione di tali oggetti. Aggiungere questa funzionalità è semplice:

```
class StringTable {
public:
```

```
  StringTable()
  { makeLogEntry("Creating
  StringTable object"); }          // aggiunta

  ~StringTable()                   // altra
  { makeLogEntry("Destroying
  StringTable object"); }          // aggiunta

   …                               // le altre funzioni come prima

private:
  std::map<int, std::string>
  values;                          // come prima
};
```

Questo sembra ragionevole, ma il fatto di dichiarare un distruttore ha un effetto collaterale potenzialmente significativo: impedisce la generazione delle operazioni di spostamento. Tuttavia, la creazione delle operazioni di copia della classe non verranno modificate. Il codice, pertanto, verrà compilato, eseguito e passerà i test funzionali. Ciò comprenderà i test della funzionalità di spostamento, poiché anche se questa classe non è più abilitata allo spostamento, le richieste di spostamento verranno compilate ed eseguite. Tali richieste provocheranno, come abbiamo detto in precedenza in questo Elemento, l'esecuzione di una copia. In pratica, il codice che "sposta" oggetti `StringTable` in realtà li copierà, ovvero copierà gli oggetti `std::map<int, std::string>` sottostanti. E la copia di una `std::map<int, std::string>` è, probabilmente, di un *ordine di grandezza* più lenta rispetto allo spostamento. Il semplice atto di aggiungere un distruttore alla classe potrebbe pertanto introdurre un significativo problema prestazionale! Se le operazioni di copia e spostamento fossero state definite esplicitamente utilizzando "**`= default`**", il problema non sarebbe sorto.

Ora, dopo questa lunga premessa sulle regole che governano le operazioni di copia e spostamento in C++11, qualcuno potrebbe iniziare a chiedersi quando rivolgerò l'attenzione alle altre due funzioni membro speciali: il costruttore standard e il distruttore. È giunto il momento, ma per una sola frase, poiché quasi nulla è cambiato in queste funzioni membro: le regole del C++11 sono praticamente le stesse del C++98.

Le regole che governano le funzioni membro speciali del C++11 sono pertanto le seguenti.

- **Costruttore standard:** stesse regole del C++98. Generato solo se la classe non contiene costruttori dichiarati dall'utente.
- **Distruttore:** sostanzialmente le stesse regole del C++98; l'unica differenza è che i distruttori sono normalmente `noexcept` (vedi Elemento 14). Come in C++98, sono virtuali solo se un distruttore nella classe base è virtuale.
- **Costruttore per copia:** stesso comportamento runtime del C++98: costruzione per copia membro a membro dei dati membro non statici. Viene generato solo se la classe non contiene un costruttore per copia dichiarato dall'utente. Viene cancellato se la classe dichiara un'operazione di spostamento. È sconsigliabile generare questa funzione in una classe con un operatore di assegnamento per copia o un distruttore dichiarato dall'utente.
- **Operatore di assegnamento per copia:** stesso comportamento runtime del C++98. Assegnamento per copia membro a membro dei dati membro non statici. Viene generato solo se la classe non contiene un operatore di assegnamento per copia dichiarato dall'utente. Viene cancellato se la classe dichiara un'operazione di spostamento. È sconsigliabile generare questa funzione in una classe con un costruttore per copia o un distruttore dichiarato dall'utente.
- **Costruttore per spostamento e operatore di assegnamento per spostamento:** entrambi svolgono lo spostamento membro a membro dei dati membro non statici. Vengono generati solo se la classe non contiene operazioni di copia, operazioni di spostamento o un distruttore dichiarati dall'utente.

Notate che le regole non parlano dell'esistenza di una funzione membro *template* che impedisce al compilatore di generare le funzioni membro speciali. Ciò significa che se il `Widget` ha il seguente aspetto,

```
class Widget {
…
  template<typename T>              // costruisce Widget
```

```
  Widget(const T& rhs);          // da qualsiasi cosa

  template<typename T>           // assegna Widget
  Widget& operator=(const T&     // da qualsiasi cosa
  rhs);
  …
};
```

i compilatori genereranno comunque delle operazioni di copia e spostamento per `Widget` (supponendo che siano esaudite le condizioni che governano la loro generazione), anche se questi template potrebbero essere istanziati per produrre le signature per il costruttore per copia e per l'operatore di assegnamento per copia (questo sarebbe il caso quando `T` è `Widget`). Ma con ogni probabilità, questo è un caso limite con cui difficilmente avrete a che fare, ma vi è un motivo per cui ne parliamo. Come vedremo nell'Elemento 26, questo fatto comporta alcune conseguenze importanti.

Argomenti da ricordare

- Le funzioni membro speciali sono quelle che i compilatori possono generare da soli: costruttore standard, distruttore, operazioni di copia e operazioni di spostamento.
- Le operazioni di spostamento vengono generate solo per le classi che non hanno operazioni di spostamento, operazioni di copia o un distruttore dichiarati esplicitamente.
- Il costruttore per copia viene generato solo per le classi che non hanno un costruttore per copia dichiarato esplicitamente e viene cancellato se viene dichiarata un'operazione di spostamento. L'operatore di assegnamento per copia viene generato solo per le classi che non hanno un operatore di assegnamento per copia dichiarato esplicitamente e viene cancellato se viene dichiarata un'operazione di spostamento. La generazione delle operazioni di copia nelle classi con un distruttore dichiarato esplicitamente è sconsigliata.
- Le funzioni membro template non sopprimono mai la generazione delle funzioni membro speciali.

1. Esistono possibilità di progettazione più flessibili: una che permette ai chiamanti di determinare se debbano essere utilizzate le parentesi o le graffe all'interno delle funzioni generate da un template. Per esempio, vedete la voce del 5 giugno 2013 in "Andrzej's C++ blog": Intuitive interface - Part I.

2. L'applicazione di `final` a una funzione virtuale impedisce che la funzione possa subire un override dalle classi derivate. `final` può essere applicato anche a una classe, nel qual caso la classe non può essere utilizzata come classe base.

3. Il controllo è normalmente piuttosto semplice. Le funzioni come `std::vector::push_back` richiamano `std::move_if_noexcept`, una variante di `std::move` che esegue condizionalmente una conversione in un rvalue (vedi Elemento 23) a seconda del fatto che il costruttore per spostamento del tipo sia `noexcept`. A sua volta, `std::move_if_noexcept` consulta `std::is_nothrow_move_constructible` e il valore di questo tipo (vedi Elemento 9) viene impiegato dal compilatore, sulla base del fatto che il costruttore per spostamento abbia una designazione `noexcept` (o `throw()`).

4. Le specifiche di interfaccia per le operazioni di spostamento sui container nella Libreria Standard non hanno `noexcept`. Tuttavia, gli implementatori possono rafforzare le specifiche per le eccezioni delle funzioni della Libreria Standard e, in realtà, è pratica comune che almeno alcune operazioni di spostamento dei container vengano dichiarate `noexcept`. Tale pratica esemplifica il consiglio fornito in questo Elemento. Avendo scoperto che è possibile scrivere operazioni di spostamento per container in modo che le eccezioni non vengano mai lanciate, gli implementatori spesso dichiarano le operazioni `noexcept`, anche se lo standard non lo richiede.

5. Le affermazioni "indipendentemente dallo stato del programma" e "senza vincoli" non legittimano i programmi il cui comportamento è già indefinito. Per esempio, `std::vector::size` ha un contratto ampio, ma questo non vuol dire che si comporterà ragionevolmente se la richiamate con un indirizzo casuale di memoria che avete convertito in uno `std::vector`. Il risultato della conversione è indefinito e pertanto non vi sono garanzie sul comportamento di un programma che contenga tale conversione.

6. Poiché `Point::xValue` restituisce `double`, anche il tipo di `mid.xValue() * 10` è `double`. I tipi in virgola mobile non possono essere utilizzati per istanziare

template o per specificare valori degli enumeratori, ma possono essere utilizzati come parte di espressioni più ampie che forniscono tipi interi. Per esempio, `static_cast<int>(mid.xValue() * 10)` può essere utilizzata per istanziare un template o per specificare il valore di un enumeratore.

4

I puntatori smart

I poeti e gli autori di canzoni hanno molto da dire sull'amore; talvolta anche sui numeri; occasionalmente su entrambi gli argomenti. Ispirati da due opere molto differenti sulle relazioni esistenti fra amore e numeri, di Elizabeth Barrett Browning ("In quanti modi ti amo? Lascia che li enumeri") e di Paul Simon ("Ci dev'essere una cinquantina di modi per lasciare qualcuno"), tentiamo di enumerare i motivi per cui non è facile amare un puntatore classico.

1. La sua dichiarazione non indica se punta a un unico oggetto o a un array di oggetti.
2. La sua dichiarazione non dice nulla sul fatto che si dovrebbe distruggere l'elemento puntato una volta che si è terminato di usarlo, ovvero se il puntatore è il *proprietario* dell'elemento a cui punta.
3. Se determinate di dover distruggere l'elemento puntato, non vi è alcun modo per sapere come. Dovreste usare `delete` o esiste un meccanismo di distruzione differente (per esempio una funzione di distruzione dedicata alla quale deve essere passato il puntatore)?
4. Se scoprite che potete distruggere l'elemento puntato come indicato al punto 1 di questo elenco, significa che potrebbe non essere possibile sapere se utilizzare la forma per un solo oggetto ("`delete`") o la forma per array ("`delete[]`"). Utilizzando la forma sbagliata, i risultati saranno

indefiniti.

5. Supponendo di esservi accertati che il puntatore è proprietario dell'elemento puntato e di aver scoperto il modo in cui è opportuno distruggerlo, è difficile garantire che la distruzione avverrà una sola volta all'interno del codice (considerando anche le distruzioni dovute alle eccezioni). Se perderemo il puntatore, avremo uno spreco di risorse e l'esecuzione di più distruzioni su un elemento può portare a comportamenti indefiniti.

6. In genere non vi è alcun modo per capire se il puntatore non è più significativo, ovvero se punta a memoria che non contiene più l'oggetto cui dovrebbe puntare il puntatore. I puntatori non significativi si hanno quando degli oggetti vengono distrutti, ma esistono ancora dei puntatori che vi puntano.

I puntatori classici, per chiarire, sono strumenti potenti, ma decenni di esperienza hanno dimostrato che basta una piccola distrazione o un'operazione non perfettamente rigorosa e questi strumenti possono mostrare tutti i loro difetti.

I *puntatori smart* (intelligenti) rappresentano un modo per risolvere questi problemi. Si tratta di *wrapper* che circondano i puntatori classici e che si comportano in generale come i puntatori classici che contengono, eliminando però molte delle loro trappole. Questo è il motivo per cui è sempre il caso di preferire i puntatori smart ai puntatori classici. I puntatori smart possono fare praticamente tutto ciò che possono fare i puntatori classici, ma con molte meno probabilità di produrre errori.

In C++11 esistono quattro puntatori smart: `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr` e `std::weak_ptr`. Sono stati concepiti per aiutare a gestire il ciclo di vita degli oggetti ad allocazione dinamica, ovvero per evitare sprechi di risorse, garantendo che tali oggetti vengano distrutti in modo appropriato e al momento appropriato (anche nel caso di eccezioni).

`std::auto_ptr` è un residuo (sconsigliato) del C++98. È stato un tentativo di standardizzare ciò che poi, in C++11, è diventato `std::unique_ptr`. L'operazione corretta richiedeva l'impiego della semantica di spostamento, assente in C++98. Per aggirare l'ostacolo, `std::auto_ptr` trasformava le operazioni di copia in spostamenti. Questo ha portato alla nascita di codice sorprendente (la copia di uno `std::auto_ptr`, gli assegna il valore nullo!) e imponeva limiti frustranti (per esempio non è possibile memorizzare dei

puntatori `std::auto_ptr` in un container).

`std::unique_ptr` fa esattamente ciò che fa `std::auto_ptr` e qualcosa in più. Il tutto in modo efficiente e senza alterare ciò che intende per "copia di un oggetto". È meglio di uno `std::auto_ptr` in ogni senso. L'unico uso residuo legittimo di `std::auto_ptr` consiste nella necessità di impiegare compilatori C++98. A meno che abbiate questo vincolo, sostituite gli `std::auto_ptr` con altrettanti `std::unique_ptr` e non pensateci più.

Le API dei puntatori smart sono molto varie. L'unica funzionalità comune a tutte è la costruzione di default. Poiché sono disponibili molte notizie sull'uso di queste API, concentrerò la mia discussione sulle informazioni che normalmente non sono presenti ove si trattano le API, ovvero sui casi d'uso degni di nota, sull'analisi dei costi runtime e così via. Queste informazioni possono fare la differenza fra il semplice uso dei puntatori smart e un uso davvero efficace dei puntatori smart.

Elemento 18 – Uso di `std::unique_ptr` per la gestione delle risorse a proprietà esclusiva

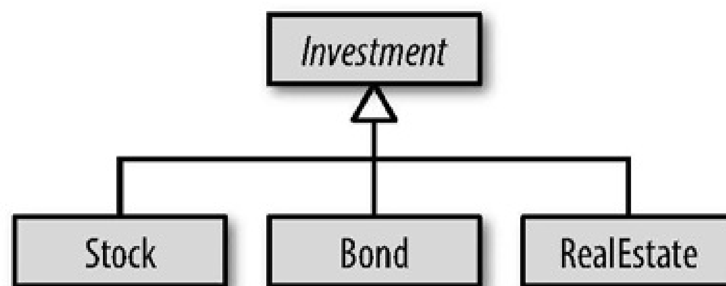
Il primo dei puntatori smart da considerare, in genere, è `std::unique_ptr`. È ragionevole presumere che, per default, gli `std::unique_ptr` siano delle stesse dimensioni dei puntatori standard e che per la maggior parte delle operazioni (compreso il deindirizzamento), essi eseguano esattamente le stesse istruzioni. Questo significa che potete usarli anche in quelle situazioni in cui non vi sia ampia disponibilità di memoria e cicli macchina. Se per queste esigenze un puntatore standard è sufficientemente compatto e veloce, lo sarà, quasi certamente, anche uno `std::unique_ptr`.

`std::unique_ptr` include la semantica della *proprietà esclusiva*. Uno `std::unique_ptr` non nullo è sempre proprietario di ciò a cui punta. Spostando uno `std::unique_ptr`, si trasferisce la proprietà dal puntatore di origine al puntatore di destinazione (il puntatore di origine viene impostato a `null`). La copia di uno `std::unique_ptr` non è consentita, poiché se poteste copiare uno `std::unique_ptr`, otterreste due `std::unique_ptr` che puntano alla stessa risorsa, ognuno dei quali pensa anche di esserne il proprietario (e, quindi, anche di poterla distruggere). `std::unique_ptr` è pertanto un *tipo move-only*. Al

momento della distruzione, uno `std::unique_ptr` non nullo distrugge la propria risorsa. Per default, la distruzione di una risorsa viene eseguita applicando `delete` al puntatore standard situato all'interno dello `std::unique_ptr`.

Un utilizzo comune di `std::unique_ptr` è come tipo restituito da una funzione factory per gli oggetti di una gerarchia. Supponete di avere una gerarchia di tipi di investimenti (per esempio, azioni, obbligazioni, immobili e così via) con classe base `Investment`.

```
class Investment { … };

class Stock:
  public Investment { … };

class Bond:
  public Investment { … };

class RealEstate:
  public Investment { … };
```



Una funzione per la creazione di tale gerarchia, in genere alloca un oggetto sullo heap e restituisce un puntatore a tale oggetto, dove il chiamante sarà responsabile della cancellazione dell'oggetto quando non ne avrà più bisogno. Questo comportamento è perfetto per `std::unique_ptr, poiché` il chiamante acquisisce la responsabilità della risorsa restituita dal produttore (ovvero la proprietà esclusiva) e lo `std::unique_ptr` cancella automaticamente l'oggetto puntato nel momento in cui esso stesso viene distrutto. Una funzione factory per la gerarchia `Investment` dovrebbe essere dichiarata nel seguente modo:

```
template<typename... Ts>            // restituisce uno
                                    std::unique_ptr
std::unique_ptr<Investment>         // a un oggetto creato
makeInvestment(Ts&&... params);     // dagli argomenti forniti
```

I chiamanti possono usare lo `std::unique_ptr` restituito nel seguente modo,

```
{
   …
   auto pInvestment =                // pInvestment è di tipo
      makeInvestment( arguments );   // std::unique_ptr<Investment>
…
}                                    // distrugge *pInvestment
```

ma potrebbero anche utilizzarlo in una situazione di migrazione della proprietà, come quando lo `std::unique_ptr` restituito dal produttore viene spostato in un container, l'elemento del container viene successivamente spostato in un dato membro di un oggetto e tale oggetto viene poi distrutto. Quando ciò accade, viene distrutto anche il dato membro `std::unique_ptr` dell'oggetto e la sua distruzione provocherà la distruzione della risorsa restituita dal produttore. Se la catena delle proprietà venisse interrotta a causa di un'eccezione o di un altro flusso di controllo atipico (per esempio, un'uscita anticipata dalla funzione o un `break` in un ciclo), lo `std::unique_ptr` proprietario della risorsa gestita richiamerebbe alla fine il proprio distruttore,[7] e la risorsa che gestiva verrebbe pertanto distrutta.

Per default, tale distruzione avverrebbe tramite `delete`, ma, durante la costruzione, gli oggetti `std::unique_ptr` possono essere configurati per utilizzare dei *cancellatori personalizzati:* funzioni (o oggetti funzione, compresi quelli che derivano dalle espressioni lambda) arbitrarie, da richiamare quando è il momento di distruggere le loro risorse. Se l'oggetto creato da `makeInvestment` non dovesse essere cancellato direttamente, ma richiedesse, prima, la scrittura di una voce sul log, `makeInvestment` potrebbe essere implementata nel seguente modo (la spiegazione segue immediatamente il codice, quindi non preoccupatevi se qualcosa non vi è chiaro).

```
auto delInvmt = [](Investment* pInvestment)   // cancellatore
        {                                      // personalizzato
            makeLogEntry(pInvestment);         // (un'espressione
            delete pInvestment;                // lambda)
        };
```

```
template<typename... Ts>                 // riveduta
std::unique_ptr<Investment,              // tipo restituito
decltype(delInvmt)>
makeInvestment(Ts&&... params)
{
  std::unique_ptr<Investment,
  decltype(delInvmt)>                    // ptr da
    pInv(nullptr, delInvmt);             // restituire
  if ( /* deve essere creato un oggetto Stock */ )
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* deve essere creato un oggetto Bond */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* deve essere creato un oggetto RealEstate */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }

  return pInv;
}
```

Tra poco ne spiegherò il funzionamento, ma innanzitutto considerate l'aspetto delle cose dal punto di vista del chiamante. Supponendo di memorizzare il risultato della chiamata a `makeInvestment` in una variabile `auto`, potete contare sul fatto che la risorsa che state utilizzando richiede un trattamento speciale durante la cancellazione. Una condizione davvero "beata", poiché con `std::unique_ptr` non dovrete preoccuparvi nel momento in cui verrà distrutta la risorsa e ancora meno del fatto che la distruzione si verifichi una e una sola volta nel corso del programma. `std::unique_ptr` si occupa automaticamente di tutte queste cose. Dal punto di vista di un client, l'interfaccia di `makeInvestment` è una vera bellezza.

Anche l'implementazione è molto comoda, una volta compreso il suo funzionamento.

- **delInvmt** è il cancellatore personalizzato per l'oggetto restituito da makeInvestment. Tutte le funzioni di cancellazione personalizzate accettano un puntatore standard all'oggetto da distruggere, poi fanno tutto ciò che è necessario per distruggere tale oggetto. In questo caso, l'azione consiste nel richiamare makeLogEntry e poi applicare delete. Usare un'espressione lambda per creare delInvmt è comodo, ma, come vedremo fra breve, è anche più efficiente rispetto a scrivere una funzione convenzionale.
- Quando deve essere utilizzato un cancellatore personalizzato, il suo tipo deve essere specificato come secondo argomento di std::unique_ptr. In questo caso, questo è il tipo di delInvmt e questo è il motivo per cui il tipo restituito da makeInvestment è std::unique_ptr<Investment, decltype(delInvmt)> (per informazioni su decltype, consultate l'Elemento 3).
- La strategia di base di makeInvestment consiste nel creare un puntatore std::unique_ptr nullo, farlo puntare a un oggetto del tipo appropriato e poi restituirlo. Per associare a pInv il cancellatore personalizzato delInvmt, lo passiamo come secondo argomento del costruttore.
- Il tentativo di assegnare un puntatore standard (per esempio da new) a std::unique_ptr non verrà accettato dal compilatore, poiché rappresenterebbe una conversione implicita da un puntatore standard a un puntatore smart. Tali conversioni implicite possono essere problematiche, pertanto i puntatori smart del C++11 non le consentono. Questo è il motivo per cui viene utilizzata reset per far sì che pInv assuma la proprietà dell'oggetto creato tramite new.
- A ogni uso di new, usiamo std::forward per eseguire il perfect-forward degli argomenti passati a makeInvestment (vedi Elemento 25). Ciò rende disponibili ai costruttori degli oggetti da creare tutte le informazioni fornite dai chiamanti.
- Il cancellatore personalizzato accetta un parametro di tipo Investment*. Indipendentemente dall'effettivo tipo dell'oggetto creato in makeInvestment (ovvero Stock, Bond o Real Estate), questo, alla fine, verrà cancellato con delete all'interno dell'espressione lambda come un oggetto Investment*. Questo significa che cancelleremo un oggetto della classe derivata tramite un puntatore alla classe base. Perché ciò funzioni, la

classe base, `Investment`, deve avere un distruttore virtuale:

```
class Investment {
public:
  …                      // componente
  virtual ~Investment(); // progettuale
  …                      // essenziale!
};
```

In C++14, grazie alla deduzione del tipo restituito dalla funzione (vedi Elemento 3), `makeInvestment` può essere implementato in questo modo, più semplice e più incapsulato:

```
template<typename... Ts>
auto makeInvestment(Ts&&... params)        // C++14
{
  auto delInvmt = [](Investment* pInvestment)  // ora questo
    {                                          // è all'interno
      makeLogEntry(pInvestment);               // di
      delete pInvestment;                      // makeInvestment
    };

  std::unique_ptr<Investment, decltype(delInvmt)>  // come
    pInv(nullptr, delInvmt);                       // prima

  if ( … )                                     // come prima
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( … )                                // come prima
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( … )                                // come prima
```

```
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;                                    // come prima
}
```

In precedenza ho fatto notare che, quando si usa il cancellatore di default (per esempio, `delete`), potete presumere ragionevolmente che gli oggetti `std::unique_ptr` siano delle stesse dimensioni dei puntatori standard. Quando inseriamo nella situazione i cancellatori personalizzati, le cose possono cambiare. I cancellatori che sono puntatori a funzioni, generalmente fanno sì che le dimensioni di uno `std::unique_ptr` crescano di una o due word. Per i cancellatori che sono oggetti funzione, le variazioni di dimensioni dipendono da quanto dello stato viene memorizzato nell'oggetto funzione. Gli oggetti funzione *stateless* (ovvero da espressioni lambda senza capture) non introducono alcun incremento dimensionale e questo significa che quando un cancellatore personalizzato può essere implementato come una funzione oppure come un'espressione lambda senza capture, è preferibile utilizzare questa seconda soluzione:

```
auto delInvmt1 = [](Investment*
pInvestment)                          // cancellatore
    {                                 // custom
      makeLogEntry(pInvestment);      // come
      delete pInvestment;             // lambda
    };                                // stateless

template<typename... Ts>              // il tipo restituito
std::unique_ptr<Investment,
decltype(delInvmt1)>                  // ha dimensioni pari a
makeInvestment(Ts&&... args);         // Investment*

void delInvmt2(Investment* pInvestment) // cancellatore
{                                     // custom
  makeLogEntry(pInvestment);          // come funzione
  delete pInvestment;
}
```

```
template<typename... Ts>                          // il tipo restituito ha
std::unique_ptr<Investment, void (*)(Investment*)> // le dimensioni di Investment*
                                                  // più almeno le dimensioni
makeInvestment(Ts&&... params);                   // di un puntatore a funzione!
```

I cancellatori che sono oggetti funzione con un ampio stato possono incrementare notevolmente le dimensioni degli oggetti `std::unique_ptr`. Se trovate che un cancellatore personalizzato aumenta le dimensioni dei vostri `std::unique_ptr` a un livello inaccettabile, probabilmente dovete cambiare approccio.

Le funzioni factory non sono l'unico caso in cui è comune impiegare `std::unique_ptr`. Sono ancora più popolari come meccanismo per implementare l'idioma PImpl. Il codice per farlo non è complesso, ma in alcuni casi è tutt'altro che semplice, per cui vi rimandiamo all'Elemento 22, dedicato a questo argomento.

Uno `std::unique_ptr` può avere due forme: una per i singoli oggetti (`std::unique_ptr<T>`) e una per gli array (`std::unique_ptr<T[]>`). Di conseguenza, non vi è mai alcuna ambiguità riguardo al tipo di entità cui punta uno `std::unique_ptr`. L'API di `std::unique_ptr` è progettata per adattarsi alla forma utilizzata. Per esempio, per la forma mono-oggetto, non esiste l'operatore di indicizzazione (`operator[]`), mentre per la forma ad array mancano gli operatori di deindirizzamento (`operator*` e `operator->`).

L'esistenza di `std::unique_ptr` per gli array dovrebbe essere di interesse puramente accademico, poiché `std::array`, `std::vector` e `std::string` sono praticamente sempre una struttura dati migliore rispetto agli array grezzi. Forse l'unica situazione in cui è utile impiegare `std::unique_ptr<T[]>` sarebbe quando si usa un'API in formato C che restituisce un puntatore standard a un array nello heap di cui si assume la proprietà.

`std::unique_ptr` è il modo C++11 di esprimere la proprietà esclusiva, ma una delle sue funzionalità più interessanti è il fatto che converte con facilità ed efficienza in uno `std::shared_ptr`:

```
std::shared_ptr<Investment> // converte uno std::unique_ptr
sp = makeInvestment(
arguments );                // in uno std::shared_ptr
```

Questo è il motivo principale per cui `std::unique_ptr` è così adatto a fungere da tipo restituito da una funzione factory. Le funzioni factory non possono sapere se i chiamanti intenderanno usare la semantica della proprietà esclusiva per l'oggetto che restituiscono o se magari sarebbe più appropriato impiegarne uno a proprietà condivisa (per esempio `std::shared_ptr`). Restituendo uno `std::unique_ptr`, le funzioni factory forniscono ai chiamanti il più efficiente puntatore smart, ma non impediscono ai chiamanti di sostituirlo con il suo parente più flessibile (per informazioni su `std::shared_ptr`, consultate l'Elemento 19).

Argomenti da ricordare

- **`std::unique_ptr`** è un puntatore smart compatto, veloce, move-only per la gestione di risorse dotato della semantica della proprietà esclusiva.
- Per default, la distruzione delle risorse avviene tramite `delete`, ma potete specificare dei cancellatori custom. I cancellatori a stati, e i puntatori a funzione usati come cancellatori, aumentano le dimensioni degli oggetti `std::unique_ptr`.
- La conversione di uno `std::unique_ptr` in uno `std::shared_ptr` è semplice.

Elemento 19 – Usare `std::shared_ptr` per la gestione delle risorse a proprietà condivisa

I programmatori che usano linguaggi dotati di meccanismi di *garbage-collection* deridono il modo in cui i programmatori C++ si preoccupano di prevenire gli sprechi di risorse. "Che primitivi!", dicono. "Sono cose che il LISP ha superato negli anni Sessanta. Sono le macchine a doversi occupare della gestione delle risorse, non gli esseri umani". Gli sviluppatori C++, a questo punto, ribattono: "Sì, sì, ma intanto tu stai parlando di una situazione in cui l'unica risorsa è la memoria e il momento in cui la risorsa viene richiamata è non deterministico. Di

gran lunga meglio la generalità e prevedibilità dei distruttori, grazie!”. Ma la nostra spavalderia è un po’ amara. Il meccanismo di garbage-collection è davvero comodo e la gestione manuale della vita degli oggetti sembra un po’ come costruire un circuito di memoria utilizzando coltelli dell’età della pietra e pelle d’orso. Perché, allora, non trarre il meglio dai due mondi: un sistema che funzioni automaticamente (come la garbage-collection), ma che si applichi a tutte le risorse e abbia un comportamento temporale prevedibile (come i distruttori)?

`std::shared_ptr` è il modo escogitato dal C++11 di unire questi due mondi. Un oggetto cui si accede tramite `std::shared_ptr` ha un ciclo di vita gestito da questi puntatori tramite una proprietà condivisa. Nessuno specifico `std::shared_ptr` ha la *proprietà dell’oggetto*. Al contrario, tutti gli `std::shared_ptr` che vi puntano collaborano per garantire la sua distruzione nel momento in cui non sarà più necessario. Quando anche l’ultimo `std::shared_ptr` che punta a un oggetto smette di puntarvi (per esempio perché lo `std::shared_ptr` viene distrutto o viene stato fatto puntare a un altro oggetto), tale `std::shared_ptr` distruggerà l’oggetto cui punta. Come nella garbage-collection, i client non devono preoccuparsi di gestire il ciclo di vita degli oggetti puntati, ma come con i distruttori, il momento in cui avviene la distruzione dell’oggetto è deterministico.

Uno `std::shared_ptr` può dire se è l’ultimo che punta a una risorsa consultando il *conteggio dei riferimenti* della risorsa stessa, un valore associato alla risorsa che registra il numero di `std::shared_ptr` che vi puntano. I costruttori di `std::shared_ptr incrementano` questo conteggio (normalmente, vedi più avanti), i distruttori di `std:: shared_ptr` decrementano questo conteggio e gli operatori di assegnamento per copia fanno entrambe le cose. Se sp1 e sp2 sono `std::shared_ptr` a oggetti differenti, l’assegnamento “`sp1 = sp2;`“ modifica `sp1` in modo che punti all’oggetto puntato da `sp2`. Il risultato pratico dell’assegnamento è che il conteggio dei riferimenti per l’oggetto originariamente puntato da `sp1` viene decrementato, mentre quello per l’oggetto puntato da `sp2` viene incrementato. Se uno `std::shared_ptr` rileva un conteggio dei riferimenti pari a zero dopo aver eseguito il proprio decremento, significa che non esistono più `std::shared_ptr` che puntano a questa risorsa e dunque lo `std::shared_ptr` la distrugge.

L’esistenza del conteggio dei riferimenti ha alcune implicazioni prestazionali.

- **Gli `std::shared_ptr` hanno dimensioni doppie rispetto ai puntatori standard**, poiché internamente contengono un puntatore standard alla risorsa e anche un puntatore standard al conteggio dei riferimenti alla risorsa.[8]
- **La memoria per il conteggio dei riferimenti deve essere allocata dinamicamente.** Teoricamente, il conteggio dei riferimenti viene associato all'oggetto puntato, il quale, però, non ne sa nulla. Pertanto non ha spazio per memorizzare il conteggio dei riferimenti (un'implicazione interessante è che in questo modo qualsiasi oggetto, anche uno di tipo standard, può essere gestito tramite puntatori `std::shared_ptr`). L'Elemento 21 spiega che il costo dell'allocazione dinamica viene evitato quando lo `std::shared_ptr` viene creato da `std::-make_shared`, ma vi sono situazioni in cui `std::make_shared` non può essere utilizzata. In ogni caso, il conteggio dei riferimenti viene memorizzato sotto forma di dati allocati dinamicamente.
- **Gli incrementi e i decrementi del conteggio dei riferimenti deve essere atomico**, poiché potrebbero esservi letture e scritture simultanee in thread differenti. Per esempio, uno `std::shared_ptr` che punta a una risorsa in un thread potrebbe eseguire il proprio distruttore (decrementando pertanto il conteggio dei riferimenti per la risorsa puntata), mentre in un altro thread uno `std::shared_ptr` allo stesso oggetto potrebbe essere copiato (incrementando pertanto lo stesso conteggio di riferimenti). Le operazioni atomiche sono generalmente più lente rispetto a quelle non atomiche, pertanto anche se i conteggi dei riferimenti hanno normalmente dimensioni pari a una sola word, potete presupporre che le operazioni di lettura e scrittura di tale valore siano relativamente costose.

Ho risvegliato la vostra curiosità quando ho scritto che i costruttori `std::shared_ptr` solo "normalmente" incrementano il conteggio dei riferimenti per l'oggetto cui puntano? La creazione di uno `std::shared_ptr` che punta a un oggetto fornisce sempre un ulteriore `std::shared_ptr` che punta all'oggetto, pertanto perché il conteggio dei riferimenti non si incrementa *sempre*?

Il motivo è l'esistenza della costruzione per spostamento. La costruzione per spostamento di uno `std::shared_ptr` da un altro `std::shared_ptr` fa sì che il `std::shared_ptr` di origine a `null` e ciò significa che nel momento in cui nasce il nuovo `std::shared_ptr`, il vecchio `std::shared_ptr` smette di puntare alla

risorsa. Come risultato, non è necessario alcun intervento sul conteggio dei riferimenti. Lo spostamento di uno `std::shared_ptr` è pertanto più veloce rispetto alla copia: la copia richiede un incremento del conteggio dei riferimenti, mentre uno spostamento no. Questo vale per l'assegnamento come per la costruzione e dunque la costruzione per spostamento è più veloce rispetto alla costruzione per copia e l'assegnamento per spostamento è più veloce rispetto all'assegnamento per copia.

Come `std::unique_ptr` (vedi Elemento 18), anche `std::shared_ptr` usa `delete` come meccanismo di default per la distruzione della risorsa, ma supporta anche i cancellatori custom. La struttura di questo supporto differisce, però, da quella di `std:: unique_ptr`. Per `std::unique_ptr`, il tipo di cancellatore fa parte del tipo del puntatore smart. Per `std::unique_ptr`, no:

```
auto loggingDel = [](Widget *pw)  // cancellatore custom
    {                             // (vedi Elemento 18)
      makeLogEntry(pw);
      delete pw;
    };

std::unique_ptr<                  // il tipo del cancellatore è
  Widget, decltype(loggingDel)    // parte del tipo del ptr
  > upw(new Widget, loggingDel);

std::shared_ptr<Widget>           // il tipo del cancellatobre non fa
  spw(new Widget, loggingDel);    // parte del tipo del ptr
```

La struttura di `std::shared_ptr` è più flessibile. Considerate due `std::shared_ptr<Widget>`, ognuno con un cancellatore custom di un tipo differente (per esempio perché i cancellatori custom sono specificati tramite espressioni lambda):

```
auto customDeleter1 = [](Widget *pw) { … };    // cancellatori custom,
auto customDeleter2 = [](Widget *pw) { … };    // ognuno di
                                               // tipo differente
std::shared_ptr<Widget> pw1(new Widget,
```

```
customDeleter1);

std::shared_ptr<Widget> pw2(new Widget,
customDeleter2);
```

Poiché `pw1` e `pw2` sono dello stesso tipo, possono essere collocati in un container di oggetti di tale tipo:

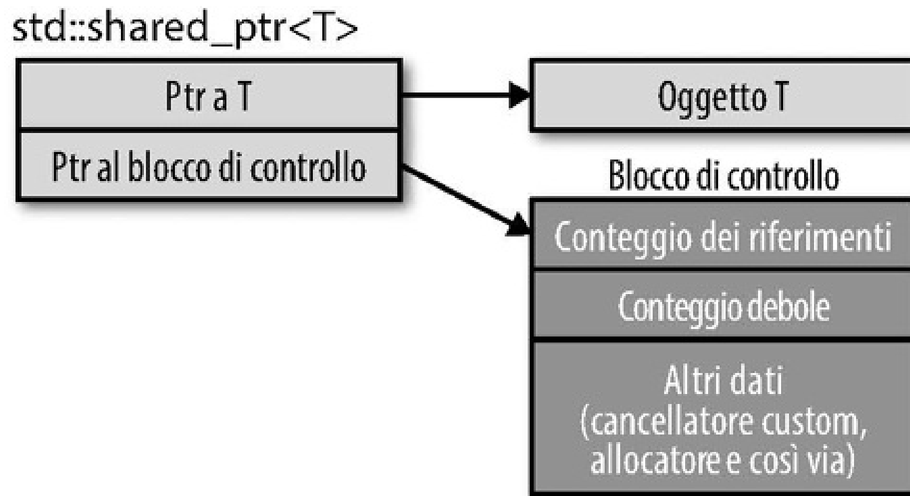
```
std::vector<std::shared_ptr<Widget>> vpw{ pwl, pw2 };
```

Possono anche essere assegnati l’uno all’altro, e possono essere loro passate delle funzioni che prendono un parametro di tipo `std::shared_ptr<Widget>`. Nessuna di queste cose può essere fatta con degli `std::unique_ptr` che differiscono per il tipo dei loro cancellatori custom, poiché il tipo del cancellatore custom determina il tipo di `std::unique_ptr`.

Un’altra differenza da `std::unique_ptr`: specificando un cancellatore custom non si cambiano le dimensioni di un oggetto `std::unique_ptr`. Indipendentemente dal cancellatore, un oggetto `std::unique_ptr` ha dimensioni pari a due puntatori. Questa è una buona cosa, ma dovrebbe anche farvi preoccupare un po’. I cancellatori custom possono essere oggetti funzione e gli oggetti funzione possono contenere quantità arbitrarie di dati. Ciò significa che le loro dimensioni possono essere arbitrariamente estese. Come può uno `std::unique_ptr` far riferimento a un cancellatore di dimensioni arbitrarie senza utilizzare altra memoria?

E infatti è così: può dover utilizzare più memoria. E tale memoria non fa parte dell’oggetto `std::shared_ptr`. Si trova nello heap o, se chi crea lo `std::shared_ptr` sfrutta il supporto per gli allocatori custom, ovunque sia situata la memoria gestita dall’allocatore. In precedenza ho detto che un oggetto `std::shared_ptr` contiene un puntatore al conteggio dei riferimenti per l’oggetto cui punta. Questo è vero, ma è leggermente fuorviante, poiché il conteggio dei riferimenti fa parte di una struttura-dati di maggiori dimensioni, chiamata *blocco di controllo*. Vi è un blocco di controllo per ogni oggetto gestito dagli `std::shared_ptr`. Il blocco di controllo contiene, oltre al conteggio dei riferimenti, una copia del cancellatore custom, qualora sia stato specificato. Se è stato specificato un’allocatore custom, allora il blocco di controllo contiene anche una copia di questo. Il blocco di controllo può anche contenere ulteriori dati, incluso, come spiega l’Elemento 21, un conteggio dei riferimenti secondario noto con il nome di conteggio debole, un argomento che però

ignoreremo in questo punto. Possiamo rappresentare la memoria associata a un oggetto `std::shared_ptr<T>` come se avesse il seguente aspetto:



Il blocco di controllo di un oggetto è configurato dalla funzione che ha creato il primo `std::shared_ptr` che rimanda all'oggetto. Sostanzialmente accade quello che si immagina dovrebbe accadere. In generale una funzione che crea uno `std::shared_ptr` verso un oggetto non può sapere se esistono altri `std::shared_ptr` che puntano già a tale oggetto, pertanto vengono utilizzate le seguenti regole per la creazione del blocco di controllo.

- **std::make_shared (vedi Elemento 21) crea sempre un blocco di controllo.** Crea sempre un nuovo oggetto, cui punta, e dunque, per certo, sa che non esiste alcun blocco di controllo per tale oggetto nel momento in cui viene richiamata.
- **Viene creato un blocco di controllo quando viene costruito uno `std::shared_ptr` da un puntatore a proprietà unica (per esempio, uno `std::unique_ptr` o uno `std::auto_ptr`).** I puntatori a proprietà unica non utilizzano blocchi di controllo, dunque in precedenza di sicuro non esisteva alcun blocco di controllo per l'oggetto puntato (nell'ambito della sua costruzione, lo `std::shared_ptr` assume la proprietà dell'oggetto puntato e pertanto il puntatore a proprietà unica viene impostato a null).
- **Quando un costruttore di `std::shared_ptr` viene richiamato con un puntatore standard, crea un blocco di controllo.** Se intendevate creare uno `std::shared_ptr` da un oggetto che ha già un blocco di controllo, presumibilmente passate come argomento del costruttore uno

std::shared_ptr o uno std::weak_ptr (vedi Elemento 20), non un puntatore standard. I costruttori di uno std::shared_ptr che prendono come argomento uno std::shared_ptr o uno std::weak_ptr non creano nuovi blocchi di controllo, poiché possono contare sui puntatori smart che vengono passati loro, i quali puntano già ai blocchi di controllo richiesti.

Una conseguenza di queste regole è il fatto che la costruzione di più std::shared_ptr da un unico puntatore standard regala un giro gratis sull'acceleratore di particelle dei comportamenti indefiniti, poiché l'oggetto puntato avrà più blocchi di controllo. Più blocchi di controllo significano più conteggi dei riferimenti e più conteggi dei riferimenti significano che l'oggetto verrà distrutto più volte (una volta per ogni conteggio dei riferimenti). Ciò significa che il codice seguente è davvero molto sbagliato:

```
auto pw = new Widget;                          // pw è un ptr standard

…

std::shared_ptr<Widget> spw1(pw, loggingDel);  // crea un blocco di controllo
                                               // per *pw
…

std::shared_ptr<Widget> spw2(pw, loggingDel);  // crea un secondo
                                               // blocco di controllo
                                               // per *pw!
```

La creazione del puntatore standard pw verso un oggetto allocato dinamicamente è sbagliata, poiché cozza contro il consiglio su cui si basa questo intero capitolo: preferire i puntatori smart rispetto ai puntatori standard (se avete dimenticato la motivazione di questo consiglio, potete rinfrescarvi la memoria leggendo quanto scritto a pagina 110). Ma, a parte questo, la riga che crea pw sarà anche un abominio stilistico, ma almeno non provoca un comportamento indefinito del programma.

Ora, il costruttore di spw1 viene richiamato con un puntatore standard, pertanto crea un blocco di controllo (e pertanto un conteggio dei riferimenti) per ciò cui

punta. In questo caso, si tratta di *pw, ovvero dell'oggetto puntato da pw. Di per sé stesso, questo non rappresenta un problema, ma poi il costruttore di spw2 viene richiamato con lo stesso puntatore standard e dunque anch'esso crea un (nuovo) blocco di controllo (e pertanto un nuovo conteggio dei riferimenti) per *pw. Alla fine, *pw ha due conteggi dei riferimenti, ognuno dei quali giungerà in qualche modo a zero e ciò, dunque, porterà a un tentativo di distruggere per due volte *pw. La seconda distruzione produrrà il comportamento indefinito.

Se ne possono trarre almeno due lezioni sull'uso degli std::shared_ptr. Innanzitutto, tentate di evitare di passare un puntatore standard a un costruttore di std::shared_ptr. L'alternativa più normale consiste nell'utilizzare std::make_shared (vedi Elemento 21), ma, nell'esempio precedente, utilizziamo dei cancellatori custom e questo non è possibile con gli std::make_shared. In secondo luogo, se dovete passare un puntatore standard a un costruttore di std::shared_ptr, passategli direttamente il risultato di new invece di passargli una variabile. Se la prima parte del codice precedente venisse riscritta nel seguente modo,

```
std::shared_ptr<Widget> spw1(new Widget,        // uso diretto di
loggingDel);                                     new
```

vi sarebbe molta meno tentazione di creare un secondo std::shared_ptr dallo stesso puntatore standard. Al suo posto, l'autore del codice che crea spw2 tenderebbe a utilizzare spw1 come argomento di inizializzazione (ovvero richiamerebbe il costruttore per copia di std::shared_ptr) e questo non porrebbe alcun problema:

```
std::shared_ptr<Widget> spw2(spw1);      // spw2 usa lo stesso
                                         // blocco di controllo di
                                         spw1
```

Un modo particolarmente sorprendente dei casi in cui l'uso di variabili puntatore standard come argomenti del costruttore di std::shared_ptr può finire per creare più blocchi di controllo, prevede l'uso del puntatore this. Supponete che il nostro programma usi degli std::shared_ptr per gestire degli oggetti Widget e che abbiamo una struttura dati che registra quanti Widget sono stati elaborati:

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Inoltre, supponete che la classe Widget abbia una funzione membro che svolge questa elaborazione:

```
class Widget {
public:
  …
  void process();
  …
};
```

Ecco un approccio dall'aspetto ragionevole per `Widget::process`:

```
void Widget::process()
{
  ...                                     // elabora il Widget
  processedWidgets.emplace_back(this);// lo aggiunge alla lista di
}                                         // Widget elaborati;
                                          // questo è sbagliato!
```

Il commento dice tutto, o almeno quasi (la parte sbagliata è il passaggio di `this`, non l'uso di `emplace_back`; se non conoscete `emplace_back`, consultate l'Elemento 42). Questo codice verrà compilato, ma passa un puntatore standard (`this`) a un container di `std::shared_ptr`. Lo `std::shared_ptr` così costruito creerà un nuovo blocco di controllo per il `Widget` (`*this`) cui punta. Non sembra troppo pericoloso, finché non si comprende che se all'esterno della funzione membro vi sono già degli `std::shared_ptr` che puntano a tale `Widget`, si ha la certezza quasi matematica di un comportamento indefinito.

L'API di `std::shared_ptr` comprende una funzionalità dedicata esattamente a questo tipo di situazione. Ha il nome probabilmente più ostico di tutta la Libreria Standard C++: `std::enable_shared_from_this`. Si tratta di un template per una classe base da cui si eredita quando si vuole che una classe gestita da `std::shared_ptr` sia in grado di creare con sicurezza uno `std::shared_ptr` a partire da un puntatore `this`. Nel nostro esempio, `Widget` erediterebbe da `std::enable_shared_from_this` nel seguente modo:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
…
void process();
```

```
…
};
```

Come ho detto, `std::enable_shared_from_this` è un template per una classe base. Il suo parametro del tipo è sempre il nome della classe derivata, pertanto `Widget` eredita da `std::enable_shared_from_this<Widget>`. Se l'idea che una classe derivata erediti da una classe base che è un template di una classe derivata vi fa venire il mal di testa, fatevelo passare. Questo codice è perfettamente lecito e l'idea su cui si basa è ben nota e ha un nome standard anche se è terribile quasi quanto `std::enable_shared_ from_this`. Il nome è *Curiously Recurring Template Pattern (CRTP)*. Se volete saperne di più, date libero sfogo al vostro motore di ricerca, poiché dovremo tornare su `std:: enable_shared_from_this`.

`std::enable_shared_from_this` definisce una funzione membro che crea uno `std:: shared_ptr` verso l'oggetto corrente, ma lo fa senza duplicare il blocco di controllo. La funzione membro è `shared_from_this` e la utilizzerete nelle funzioni membro ogni volta che vorrete uno `std::shared_ptr` che punti allo stesso oggetto del puntatore `this`. Ecco un'implementazione sicura di `Widget::process`:

```
void Widget::process()
{
  // come prima, elabora il Widget
  …
  // aggiunge std::shared_ptr all'oggetto corrente
  processedWidgets.emplace_back(shared_from_this());
}
```

Internamente, `shared_from_this` ricerca il blocco di controllo dell'oggetto corrente e crea un nuovo `std::shared_ptr` che fa riferimento a tale blocco di controllo. Il meccanismo conta sul fatto che l'oggetto corrente abbia già un proprio blocco di controllo. Perché sia così, vi deve già essere uno `std::shared_ptr` (per esempio all'esterno della funzione membro che richiama `std::shared_ptr`) che punta all'oggetto corrente. Se tale `std::shared_ptr` non esiste (ovvero, se all'oggetto corrente non è associato alcun blocco di controllo), il comportamento è indefinito, sebbene in genere `shared_from_ this` lanci un'eccezione.

Per evitare che i client richiamino delle funzioni membro che richiamano `shared_ from_this` prima che uno `std::shared_ptr` punti all'oggetto, le classi che ereditano da `std::enable_shared_from_this` spesso dichiarano i propri

costruttori `private` e fanno in modo che i client creino gli oggetti richiamando le funzioni factory che restituiscono `std::shared_ptr`. `Widget`, per esempio potrebbe avere seguente aspetto:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  // funzione factory che esegue il perfect-forward
  // degli argomenti a un costruttore privato
  template<typename... Ts>
  static std::shared_ptr<Widget> create(Ts&&... params);

  …
  void process();      // come prima
  …

private:
  …                    // costruttori
};
```

Al momento, potreste ricordare solo vagamente che la nostra discussione sui blocchi di controllo era in realtà motivata da un desiderio di comprendere i costi associati agli `std::shared_ptr`. Ora che sappiamo come evitare la creazione di troppi blocchi di controllo, torniamo all'argomento originario.

Un blocco di controllo ha, generalmente, dimensioni pari solo a poche word, anche se i cancellatori e gli allocatori custom possono aumentarle. L'implementazione consueta del blocco di controllo è più sofisticata di quello che potreste immaginare. Utilizza l'ereditarietà e vi è perfino una funzione virtuale (utilizzata per garantire che l'oggetto puntato venga distrutto correttamente). Ciò significa che l'uso di `std::shared_ptr` incorre anche nel costo del meccanismo della funzione virtuale utilizzata dal blocco di controllo.

Ora che sapete dei blocchi di controllo allocati dinamicamente, dei cancellatori e degli allocatori di dimensioni arbitrariamente estese, del meccanismo della funzione virtuale e delle manipolazioni atomiche del conteggio dei riferimenti, il vostro entusiasmo per gli `std::shared_ptr` potrebbe essersi ormai dissolto. Questo è positivo.

Essi non rappresentano la soluzione migliore per ogni problema di gestione delle risorse. Ma per la funzionalità che forniscono, gli `std::shared_ptr` hanno un costo molto ragionevole. In condizioni tipiche, quando vengono utilizzati il cancellatore e l'allocatore di default e quando lo `std::shared_ptr` viene creato

da `std::make_shared`, il blocco di controllo ha dimensioni di sole tre word e la sua allocazione è sostanzialmente banale (viene incorporata nell'allocazione della memoria dell'oggetto puntato; per dettagli, vedi Elemento 21). Il deindirizzamento di uno `std::shared_ptr` non è più costoso del deindirizzamento di un puntatore standard. L'esecuzione di un'operazione che richiede la manipolazione del conteggio dei riferimenti (ovvero la costruzione per copia, l'assegnamento per copia o la distruzione) prevede una o due operazioni atomiche, ma queste operazioni si riducono generalmente a singole istruzioni macchina e pertanto, anche se possono essere costose rispetto alle istruzioni non atomiche, si tratta comunque di singole istruzioni. Il meccanismo della funzione virtuale nel blocco di controllo viene generalmente utilizzato una sola volta per l'oggetto gestito da `std::shared_ptr`: quando l'oggetto viene distrutto.

In cambio di questo costo tutto sommato modesto, si ottiene la gestione automatica del ciclo di vita delle risorse allocate dinamicamente. Nella maggior parte dei casi, l'uso di `std::shared_ptr` è decisamente preferibile rispetto al tentativo di gestire manualmente il ciclo di vita di un oggetto con proprietà condivisa. Se pensate di non potervi permettere l'uso di `std::shared_ptr`, riconsiderate se avete davvero bisogno della proprietà condivisa. Se *potete farcela* con la proprietà esclusiva, `std::unique_ptr` rappresenta una scelta migliore. Il suo profilo prestazionale è analogo a quello dei puntatori standard e l'"aggiornamento" da `std::unique_ptr` a `std::shared_ptr` è semplice, poiché può essere creato a partire da uno `std::unique_ptr`.

Non vale il contrario. Una volta che avete avviato la gestione del ciclo di vita di una risorsa con uno un `std::shared_ptr`, non potete cambiare idea. Anche se il conteggio dei riferimenti è uno solo, non potete reclamare la proprietà della risorsa per gestirla tramite uno `std::unique_ptr` (per fare un esempio). Il contratto di proprietà fra una risorsa e gli `std::shared_ptr` che vi puntano è del tipo "finché morte non ci separi. Niente divorzio, niente annullamento, nessuna dispensa".

Un'altra cosa che gli `std::shared_ptr` non possono fare è lavorare con gli array. Un'ulteriore differenza da `std::unique_ptr: std::shared_ptr` ha un'API che è progettata solo per puntatori che rimandano a singoli oggetti. Non esiste `std::shared_ptr<T[]>`. Di tanto in tanto, i programmatori più "avanti" saltano fuori con l'idea di utilizzare uno `std::shared_ptr<T>` per puntare a un array, specificando un cancellatore custom per eseguire la cancellazione

dell’array (in pratica `delete[]`). La cosa può anche risultare compilabile, ma è un’idea terribile. Innanzitutto, `std::shared_ptr` non offre alcun `operator[]`, pertanto l’indicizzazione dell’array richiede complessissime espressioni basate sull’aritmetica dei puntatori. Inoltre `std::shared_ptr` supporta le conversioni di puntatore da-derivata-a-base, che hanno senso per singoli oggetti ma provocano enormi problemi nella gestione dei tipi se il tutto viene applicato agli array (per questo motivo, l’API di `std::unique_ptr<T[]>` proibisce tali conversioni). Ma soprattutto, data la varietà di alternative offerte dal C++11 per gli array (ovvero `std::array, std::vector e std::string`), la dichiarazione di un puntatore smart a un array non-smart è quasi sempre sintomo di cattiva progettazione.

Argomenti da ricordare

- **`std::shared_ptr`** offre un comodo approccio di tipo garbage-collection alla gestione condivisa del ciclo di vita di risorse arbitrarie.
- Rispetto a `std::unique_ptr`, gli oggetti `std::shared_ptr` hanno dimensioni generalmente doppie, aumentando pertanto le dimensioni dei blocchi di controllo e obbligando a manipolazioni atomiche del conteggio dei riferimenti.
- La distruzione standard delle risorse avviene tramite `delete`, ma sono supportati i cancellatori custom. Il tipo di cancellatore non ha alcun effetto sul tipo di `std::shared_ptr`.
- Evitate di creare `std::shared_ptr` da quelle variabili il cui tipo è un puntatore standard.

Elemento 20 – Usare `std::weak_ptr` per puntatori di tipo `std::shared_ptr` che possono “pendere”

Paradossalmente, può essere comodo avere un puntatore smart che si comporta come uno `std::shared_ptr` (vedi Elemento 19), ma che non partecipa alla proprietà condivisa della risorsa puntata. In altre parole, un puntatore come `std::shared_ptr` che non influenzi il conteggio dei riferimenti di un oggetto.

Questo tipo di puntatore smart deve affrontare un problema sconosciuto agli `std::shared_ptr`: la possibilità che ciò cui punta sia stato distrutto. Un puntatore smart "completo" gestirebbe questo problema controllando quando è "pendente", ovvero quando l'oggetto cui dovrebbe puntare non esiste più. Questo è esattamente lo scopo del puntatore smart `std::weak_ptr`.

Potreste chiedervi che utilità abbia uno `std::weak_ptr`. I vostri dubbi possono anche crescere esaminando l'API dello `std::weak_ptr`. Sembra tutt'altro che "smart". Gli `std::weak_ptr` non possono essere dereferenziati, né è possibile controllare se sono `null`. Questo perché uno `std::weak_ptr` non è un puntatore smart indipendente. È un'estensione di `std::shared_ptr`.

La relazione inizia alla nascita. Uno `std::weak_ptr` viene generalmente creato a partire da uno `std::shared_ptr`. Punta allo stesso luogo dello `std::shared_ptr` usato per inizializzarlo, ma non influenza il conteggio dei riferimenti all'oggetto puntato:

```
auto spw =                        // dopo che spw è costruito,
  std::make_shared<Widget>();     // il conteggio dei riferimenti (RC)
                                  // del Wiget puntato è 1.
                                  // (Vedi Elemento 21 per info su
                                  // std::make_shared.)

  …

  std::weak_ptr<Widget> wpw(spw); // wpw punta allo stesso Widget
                                  // di spw. RC rimane 1
  …

  spw = nullptr;                  // RC va a 0, e il
                                  // Widget viene distrutto.
                                  // Ora wpw è pendente
```

I puntatori `std::weak_ptr` che si trovano in uno stato "pendente" sono come *scaduti*. La cosa può essere verificata direttamente,

```
if (wpw.expired()) …          // se wpw non punta
                              // a un oggetto…
```

ma spesso si ha bisogno di controllare se uno `std::weak_ptr` è scaduto e, se non lo è (ovvero se non è "pendente"), di accedere all'oggetto cui punta. Più facile a dirsi che a farsi! Poiché gli `std::weak_ptr` non hanno operazioni di deindirizzamento, non vi è alcun modo per scrivere il codice. Anche se vi fosse, il fatto di separare il controllo e il deindirizzamento introdurrebbe una situazione di competizione: fra la chiamata a `expired` e l'azione di deindirizzamento, un altro thread potrebbe riassegnare o distruggere l'ultimo `std::shared_ptr` che punta all'oggetto, provocando la distruzione di tale oggetto. In tal caso, l'operazione di deindirizzamento avrebbe un comportamento indefinito.

È necessario ricorrere a un'operazione atomica che controlla se lo `std::weak_ptr` è scaduto e, in caso contrario, offre l'accesso all'oggetto cui punta. Questo controllo può essere fatto creando uno `std::shared_ptr` dallo `std::weak_ptr`. L'operazione può avere due forme, a seconda di ciò che volete che succeda se lo `std::weak_ptr` è scaduto quando tentate di usarlo per creare uno `std::shared_ptr`. Una forma è `std::weak_ptr::lock`, che restituisce uno `std::shared_ptr`. Se lo `std::weak_ptr` è scaduto, lo `std::shared_ptr` è `null`:

```
std::shared_ptr<Widget> spw1 =
wpw.lock();                          // se wpw è scaduto,
                                     // spw1 è null

auto spw2 = wpw.lock();              // come sopra,
                                     // but uses auto
```

L'altra forma è il costruttore di `std::shared_ptr` che accetta come argomento uno `std::weak_ptr`. In questo caso, se lo `std::weak_ptr` è scaduto, viene lanciata un'eccezione:

```
std::shared_ptr<Widget>
spw3(wpw);                     // se wpw è scaduto,
                               // lancia std::bad_weak_ptr
```

Ma probabilmente vi state ancora chiedendo a cosa possano servire gli `std::weak_ptr`. Considerate una funzione factory che produca dei puntatori

smart a oggetti di sola lettura basati su un ID univoco. In base al consiglio dell'Elemento 18 riguardante i tipi restituiti dalle funzioni factory, questa restituisce uno `std::unique_ptr`:

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

Se `loadWidget` è una chiamata costosa (per esempio perché svolge operazioni di I/O su file o database) ed è comune che gli ID vengano utilizzati ripetutamente, un'ottimizzazione ragionevole consisterebbe nello scrivere una funzione che fa ciò che fa `loadWidget`, ma che salva in cache i risultati. Generare un'area temporanea per ogni `Widget` richiesto può portare a problemi prestazionali, ma un'altra ottimizzazione ragionevole consisterebbe nel distruggere i `Widget` temporanei quando non sono più necessari.

Per questa funzione factory dotata di gestione cache, non sarebbe appropriato usare come tipo restituito uno `std::unique_ptr`. I chiamanti dovranno certamente ricevere dei puntatori smart agli oggetti temporanei e dovranno certamente determinare il ciclo di vita di questi oggetti, ma anche la cache richiede un puntatore agli oggetti. I puntatori della cache devono essere in grado di rilevare quando sono pendenti, poiché quando i client factory terminano di utilizzare un oggetto restituito dalla factory, tale oggetto verrà distrutto e la voce corrispondente nella cache risulterà pendente. I puntatori della cache dovranno pertanto essere puntatori `std::weak_ptr`, ovvero essere in grado di rilevare quando sono pendenti. Ciò significa che il tipo restituito dalla factory dovrebbe essere uno `std::shared_ptr`, poiché gli `std::weak_ptr` possono rilevare quando sono pendenti solo quando il ciclo di vita di un oggetto viene gestito da uno `std::shared_ptr`.

Ecco l'implementazione banale della versione cache di `loadWidget`:

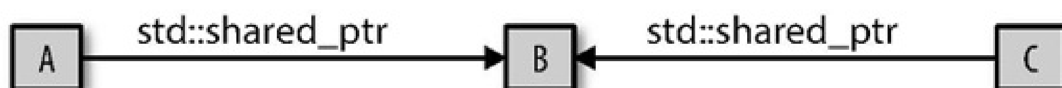
```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
  static std::unordered_map<WidgetID,
         std::weak_ptr<const Widget>> cache;

  auto objPtr = cache[id].lock();   // objPtr è uno std::shared_ptr
                                    // all'oggetto in cache (or nullo
                                    // se l'oggetto non è nella cache)
```

```
  if (!objPtr) {             // se non è nella cache,
    objPtr =
    loadWidget(id);          // caricalo
    cache[id] = objPtr;      // e mettilo in cache
  }
  return objPtr;
}
```

Questa implementazione impiega uno dei container ad hash table C++11 (`std::unordered_map`), anche se non offre le funzioni di hash e di confronto dell'uguaglianza di `WidgetID`, che dovrebbero essere presenti.

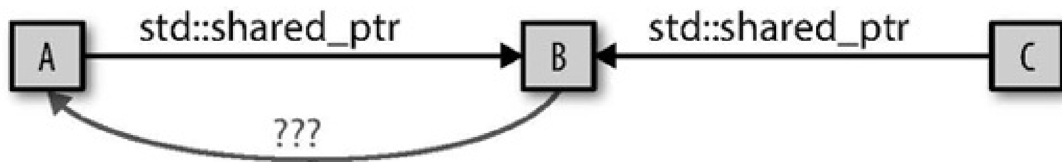
L'implementazione di `fastLoadWidget` ignora il fatto che la cache può accumulare `std::weak_ptr` scaduti corrispondenti ai `Widget` che non sono più in uso (perché sono stati distrutti). L'implementazione può essere migliorata, ma anziché perdere tempo su un problema che non darebbe alcuna conoscenza del funzionamento degli `std::weak_ptr`, consideriamo un secondo caso d'uso: lo schema dell'osservatore. I componenti principali di questo schema sono i soggetti (quegli oggetti il cui stato può cambiare) e gli osservatori (quegli oggetti cui viene notificato un cambio di stato). Nella maggior parte delle implementazioni, ogni soggetto contiene un dato membro che, a sua volta, contiene dei puntatori ai propri osservatori. Ciò consente ai soggetti di emettere notifiche di cambiamento di stato. I soggetti non sono interessati a controllare il ciclo di vita dei loro osservatori (per esempio, quando vengono distrutti), ma sono invece molto interessati ad assicurarsi che se un osservatore viene distrutto, i soggetti non tentino, poi, di accedervi. Una progettazione ragionevole è che ogni soggetto contenga un container di `std::weak_ptr` che punta ai suoi osservatori; ciò consente al soggetto di determinare se un puntatore è pendente, prima di usarlo.

Come esempio finale dell'utilità degli `std::weak_ptr`, considerate una struttura dati contenente gli oggetti `A`, `B` e C, dove `A` e `C` hanno la proprietà condivisa di `B` e pertanto contengono degli `std::shared_ptr` verso `B`:



Supponete che sia utile anche avere un puntatore che da `B` riporti ad `A`. Quale tipo

di puntatore sarebbe?



Vi sono tre scelte.

- **Un puntatore standard.** Con questo approccio, se `A` venisse distrutto, ma `C` continuasse a puntare a `B`, `B` conterrà un puntatore pendente ad `A`. `B` non sarebbe in grado di rilevarlo e pertanto potrebbe inavvertitamente deindirizzare il puntatore pendente. Ciò porterebbe a un comportamento indefinito.
- **Uno `std::shared_ptr`.** In tal modo, `A` e `B` contengono degli `std::shared_ptr` l'uno all'altro. Il ciclo `std::shared_ptr` risultante (`A` punta a `B` e `B` punta a `A`) impedirebbe la distruzione sia di `A` sia di `B`. Anche se `A` e `B` fossero irraggiungibili da parte di altre strutture dati del programma (per esempio perché `C` non punta più a `B`), ognuno di essi avrà un conteggio dei riferimenti pari a `1`. Se ciò dovesse accadere, `A` e `B` sarebbero inutilizzabili e perderebbero ogni scopo pratico: sarebbe impossibile per il programma accedervi, ma le loro risorse non verrebbero mai reclamate.
- **Uno `std::weak_ptr`.** Questo eviterà entrambi i problemi precedenti. Se `A` venisse distrutto, il puntatore ad `A` contenuto in `B` sarebbe pendente, ma `B` sarebbe in grado di rilevarlo. Inoltre, anche se `A` e `B` puntano l'uno all'altro, il puntatore di `B` non influenzerà il conteggio dei riferimenti di `A`, il che non impedirà pertanto la distruzione di `A` nel momento in cui non vi sono più `std::shared_ptr` che vi puntano.

`std::weak_ptr` rappresenta chiaramente la scelta migliore. Tuttavia è il caso di notare che la necessità di impiegare degli `std::weak_ptr` per spezzare i cicli di `std::shared_ptr` non è molto comune. È una struttura dati fortemente gerarchica, come gli alberi, dove i nodi figli sono tipicamente di proprietà solo dei loro genitori. Quando un nodo genitore viene distrutto, anche i suoi nodi figli dovrebbero essere distrutti. Il link dai genitori ai figli sono pertanto rappresentati al meglio da `std::unique_ptr`. Il link di ritorno dai figli ai genitori possono essere implementati in sicurezza tramite puntatori standard, poiché un nodo

figlio non dovrà mai avere un ciclo di vita che si estende oltre quello del proprio genitore. Pertanto non vi è alcun rischio che un nodo figlio deindirizzi un puntatore pendente al genitore.

Naturalmente, non tutte le strutture-dati basate su puntatori sono strettamente gerarchiche e in questo caso (e anche nelle situazioni come il caching e l'implementazione delle liste di osservatori), è bello sapere di poter contare sugli `std::weak_ptr`.

Dal punto di vista dell'efficienza, il comportamento degli `std::weak_ptr` è sostanzialmente lo stesso degli `std::shared_ptr`. Gli oggetti `std::weak_ptr` hanno le stesse dimensioni degli oggetti `std::shared_ptr`, utilizzano gli stessi blocchi di controllo degli `std::shared_ptr` (vedi Elemento 19) e operazioni come la costruzione, la distruzione e l'assegnamento prevedono manipolazioni atomiche del conteggio dei riferimenti. Questo probabilmente vi sorprenderà, poiché ho scritto all'inizio di questo punto che gli `std::weak_ptr` non partecipano al conteggio dei riferimenti. In realtà questo non è esattamente ciò che ho scritto. Ciò che ho scritto è che gli `std::weak_ptr` non partecipano alla *proprietà condivisa* degli oggetti e pertanto non influenzano il *conteggio dei riferimenti degli oggetti cui puntano*. Esiste in realtà un secondo conteggio dei riferimenti nel blocco di controllo ed è proprio questo secondo conteggio dei riferimenti che viene manipolato dagli `std::weak_ptr`. Per i dettagli, basta proseguire la lettura con l'Elemento 21.

Argomenti da ricordare

- Usate degli `std::weak_ptr` per i puntatori di tipo `std::shared_ptr` che possono "pendere".
- Fra i potenziali casi d'uso degli `std::weak_ptr` vi sono la gestione cache, le liste di osservatori e le situazioni in cui occorre evitare il formarsi di cicli di `std::shared_ptr`.

Elemento 21 – Usare `std::make_unique` e `std::make_shared` per gestire l'uso di new

Iniziamo a chiarire cosa sono le funzioni `std::make_unique` e `std::make_shared`. `std::make_shared` fa parte del C++11, ma, purtroppo, `std::make_unique` no. È entrata nella Libreria Standard a partire dal C++14. Se utilizzate il C++11, non temete, poiché potete realizzare una versione semplificata di `std::make_unique` con una certa facilità. Eccola qui:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
  return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

Come potete vedere, `make_unique` non fa altro che eseguire un perfect-forward dei propri parametri al costruttore dell'oggetto, costruire uno `std::unique_ptr` dal puntatore standard prodotto da `new` e restituire lo `std::unique_ptr` così creato. Questa forma della funzione non supporta gli array o i cancellatori custom (vedi Elemento 18), ma mostra come creare, con poca fatica, una `make_unique` se ne avete bisogno.[9] Ricordatevi solo di non inserire la vostra versione nel namespace `std`, poiché potrebbe entrare in conflitto con la versione standard quando eseguirete l'aggiornamento all'implementazione C++14 della Libreria Standard.

`std::make_unique` e `std::make_shared` sono due delle tre *funzioni make*: funzioni che accettano un numero arbitrario di argomenti, li inoltrano al costruttore per ottenere un oggetto allocato dinamicamente e restituiscono un puntatore smart a tale oggetto. La terza funzione `make` è `std::allocate_shared`, che si comporta un po' come `std::make_shared`, tranne per il fatto che il suo primo argomento è un oggetto allocatore da utilizzare per l'allocazione dinamica della memoria.

Considerando la creazione di puntatori smart, anche il confronto più banale fra utilizzo o meno di una funzione `make` rivela il motivo principale per cui è preferibile utilizzare tali funzioni. Considerate:

```
auto upw1(std::make_unique<Widget>());   // con funzione make

std::unique_ptr<Widget> upw2(new Widget);   // senza

auto spw1(std::make_shared<Widget>());   // con funzione make
```

```
std::shared_ptr<Widget> spw2(new Widget);          // senza
```

Ho evidenziato la differenza essenziale: le versioni che usano `new` devono ripetere il tipo da creare, mentre le funzioni `make` no. La ripetizione del tipo si scontra contro uno dei pilastri dell'ingegneria del software: la duplicazione del codice deve essere evitata. La duplicazione nel codice sorgente aumenta i tempi di compilazione, può portare alla creazione di codice oggetto debole e, in generale, complica la manutenibilità della base di codice. Spesso, poi, evolve in codice incoerente e l'incoerenza nel codice base spesso porta all'insorgere di bug. Fra l'altro, scrivere le cose due volte richiede il doppio del tempo e chi, fra i programmatori, non cerca in tutti i modi di ridurre l'impegno di digitazione? Il secondo motivo per preferire le funzioni `make` ha a che fare con la sicurezza delle eccezioni. Supponete di avere una funzione per elaborare un `Widget` sulla base di una certa priorità:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

Il passaggio dello `std::shared_ptr` per valore può sembrare sospetto, ma l'Elemento 41 spiega che se `processWidget` esegue sempre una copia dello `std::shared_ptr` (per esempio memorizzandolo in una struttura dati che tiene traccia dei `Widget` che sono stati elaborati), questa può essere una scelta progettuale ragionevole.

Ora, supponete di avere una funzione per calcolare la priorità,

```
int computePriority();
```

e di utilizzarla in una chiamata a `processWidget` che utilizza `new` al posto di `std::make_shared`:

```
processWidget(std::shared_ptr<Widget>          // potenziale
(new Widget),
computePriority());                            // spreco
                                               // di risorse!
```

Come indica il commento, a causa del `new` questo codice potrebbe sprecare il `Widget`. Ma in quale modo? Sia il codice chiamante sia la funzione chiamata

usano degli `std::shared_ptr`, che sono progettati per prevenire lo spreco di risorse. Distruggono automaticamente l'oggetto cui puntano nel momento in cui se ne va anche l'ultimo `std:: shared_ptr`. Se tutti usano ovunque degli `std::shared_ptr`, come si può produrre questo spreco?

La risposta ha che fare con la traduzione del codice sorgente in codice oggetto effettuata dai compilatori. Runtime, gli argomenti di una funzione devono essere valutati prima che la funzione possa essere richiamata; pertanto, nella chiamata a `processWidget`, prima che `processWidget` possa iniziare la propria esecuzione devono verificarsi le seguenti attività.

- Deve essere valutata l'espressione `new Widget`, ovvero nello heap deve essere creato un `Widget`.
- Deve essere eseguito il costruttore dello `std::shared_ptr<Widget>` responsabile della gestione del puntatore prodotto da `new`.
- Deve essere attiva `computePriority`.

I compilatori non sono obbligati a generare codice che svolge queste attività esattamente in quest'ordine. `new Widget` deve necessariamente essere eseguita prima di richiamare il costruttore di `std::shared_ptr`, poiché il risultato di tale `new` viene poi utilizzato come argomento del costruttore, ma `compute Priority` può essere eseguita prima, dopo o anche *fra* queste due chiamate. In pratica, i compilatori possono produrre codice per eseguire le operazioni nel seguente ordine.

1. Esegui `new Widget`.
2. Esegui `computePriority`.
3. Lancia il costruttore di `std::shared_ptr`.

Se viene generato questo codice e, runtime, `computePriority` produce un'eccezione, il `Widget` allocato dinamicamente nel Passo 1 si perderà, poiché non verrà mai memorizzato nello `std::shared_ptr` che dovrebbe iniziare a gestirlo nel Passo 3.

L'uso di `std::make_shared` evita questo problema. Il codice chiamante avrebbe il seguente aspetto:

```
processWidget(std::make_shared<Widget>
(),                                          // nessun potenziale
```

```
    computePriority());               // spreco di risorse
```

Runtime, verranno richiamate per prime o std::make_shared o computePriority. Nel primo caso, il puntatore standard al Widget allocato dinamicamente viene memorizzato con sicurezza nello std::shared_ptr restituito prima che venga richiamata computePriority. Se poi computePriority lancia un'eccezione, il distruttore di computePriority vedrà che lo std::shared_ptr di cui è proprietario è distrutto. E se per prima viene richiamata computePriority e questa lancia l'eccezione, std::make_shared non verrà richiamata e pertanto non vi sarà alcun Widget ad allocazione dinamica di cui preoccuparsi.

Se sostituiamo std::shared_ptr e std::make_shared con std::unique_ptr e std::make_unique, si applica esattamente lo stesso ragionamento. Utilizzare std::make_unique invece di new è pertanto altrettanto importante nella scrittura di codice sicuro per le eccezioni che utilizzare std::make_shared.

Una funzionalità speciale di std::make_shared (rispetto all'utilizzo diretto di new) è la maggiore efficienza. std::make_shared consente ai compilatori di generare codice più compatto e veloce, che impiega strutture dati più snelle. Considerate il seguente uso diretto di new:

```
std::shared_ptr<Widget> spw(new Widget);
```

È ovvio che questo codice preveda un'allocazione di memoria, ma in realtà ne svolge due. L'Elemento 19 spiega che ogni std::shared_ptr ha un blocco di controllo che contiene, fra le altre cose, il conteggio dei riferimenti all'oggetto puntato. La memoria di questo blocco di controllo viene allocata nel costruttore di std::shared_ptr. L'utilizzo diretto di new, pertanto, richiede un'allocazione di memoria per il Widget e una seconda allocazione di memoria per il blocco di controllo.

Se invece si usa std::make_shared,

```
auto spw = std::make_shared<Widget>();
```

basta un'allocazione. Questo perché std::make_shared alloca un unico frammento di memoria che contiene sia l'oggetto Widget sia il blocco di controllo. Questa ottimizzazione riduce le dimensioni statiche del programma, poiché il codice contiene una sola chiamata di allocazione della memoria e

aumenta anche la velocità del codice eseguibile, in quanto la memoria viene allocata una sola volta. Inoltre, l'uso di `std::make_shared` evita la necessità di conservare alcune informazioni di gestione nel blocco di controllo, riducendo potenzialmente l'occupazione di memoria da parte del programma.

L'analisi dell'efficienza per `std::make_shared` è ugualmente applicabile anche a `std::allocate_shared`, e pertanto i vantaggi prestazionali di `std::make_shared` si estendono anche a tale funzione.

Gli argomenti per preferire le funzioni `make` rispetto all'utilizzo diretto di `new` sono davvero notevoli. Ma nonostante tutti i loro indubbi vantaggi in termini di ingegnerizzazione del software, sicurezza di gestione delle eccezioni ed efficienza, il consiglio di questo Elemento è quello di *preferire* le funzioni `make`, non quello di contare esclusivamente su di esse. Questo perché vi sono casi in cui esse non possono o non devono essere impiegate.

Per esempio, nessuna delle funzioni `make` permette di specificare dei cancellatori custom (vedere gli Elementi 18 e 19), mentre sia `std::unique_ptr` sia `std::shared_ptr` hanno dei costruttori che li consentono. Dato un cancellatore custom per un oggetto,

```
auto widgetDeleter = [](Widget* pw) { … };
```

creare un puntatore smart che lo utilizza è molto semplice con `new`:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
  upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

Al contrario, non esiste alcun modo per fare la stessa cosa con una funzione `make`.

Un secondo limite delle funzioni `make` deriva da un dettaglio sintattico della loro implementazione. L'Elemento 7 spiega che quando si crea un oggetto il cui tipo esegue l'overloading di costruttori sia con sia senza parametri `std::initializer_list`, la creazione di un oggetto utilizzando parentesi graffe preferisce il costruttore `std::initializer_list`, mentre la creazione di un oggetto utilizzando le normali parentesi preferisce il costruttore non-`std::initializer_list`. Le funzioni `make` eseguono un perfect-forward dei propri parametri al costruttore dell'oggetto, ma lo fanno utilizzando parentesi o

graffe? Per alcuni tipi, la risposta a questa domanda introduce una grande differenza. Per esempio, nelle seguenti chiamate,

```
auto upv = std::make_unique<std::vector<int>>(10, 20);

auto spv = std::make_shared<std::vector<int>>(10, 20);
```

i puntatori smart risultanti puntano a uno `std::vector` di 10 elementi, ognuno dei quali ha valore 20 oppure a uno `std::vector` di 2 elementi, uno dei quali vale 10 e l'altro 20? O forse il risultato è indeterminato?

L'aspetto positivo è che il risultato non è indeterminato: entrambe le chiamate creano degli `std::vector` di 10 elementi tutti uguali a 20. Ciò significa che all'interno delle funzioni `make`, il codice di perfect-forward utilizza le parentesi, non le graffe. Il problema è che se volete costruire il vostro oggetto puntato utilizzando un inizializzatore a graffe, dovete utilizzare direttamente `new`. L'utilizzo di una funzione `make` richiederebbe la possibilità di eseguire il perfect-forward di un inizializzatore a graffe, ma, come spiega l'Elemento 30, gli iniziatori a graffe non possono essere inoltrati con un perfect-forward. Tuttavia, l'Elemento 30 descrive anche una soluzione: utilizzare la deduzione del tipo `auto` per creare un oggetto `std::initializer_list` a partire da un inizializzatore a graffe (vedi Elemento 2), poi passare l'oggetto creato con `auto` attraverso la funzione `make`:

```
// crea std::initializer_list
auto initList = { 10, 20 };

// crea std::vector usando il costr. di std::initializer_list
auto spv = std::make_shared<std::vector<int>>(initList);
```

Per `std::unique_ptr`, queste due situazioni (cancellatori custom e inizializzatori a graffe) sono le uniche in cui le funzioni `make` sono problematiche. Per `std::shared_ptr` e le sue funzioni `make`, ve ne sono altre due. Si tratta di due casi limite, ma alcuni sviluppatori operano proprio a cavallo di questi limiti e voi potreste essere uno di loro.

Alcune classi definiscono le loro versioni di `operator new` e `operator delete`. La presenza di queste funzioni implica che le routine di allocazione e deallocazione della memoria globale per gli oggetti di questi tipi sono inappropriate. Spesso vengono progettate delle routine specifiche per la classe,

unicamente per allocare e deallocare frammenti di memoria di dimensioni adatte agli oggetti di questa classe; per esempio, `operator new` e `operator delete` per la classe `Widget` vengono frequentemente progettati unicamente per gestire l'allocazione e deallocazione di frammenti di memoria di dimensioni esattamente pari a `sizeof(Widget)`. Queste routine sono poco adatte al supporto di `std::shared_ptr` per l'allocazione custom (tramite `std:: allocate_shared`) e deallocazione custom (tramite cancellatori custom), poiché la quantità di memoria richiesta da `std::allocate_shared` non è delle dimensioni dell'oggetto allocato dinamicamente, ma delle dimensioni di tale oggetto *più* le dimensioni di un blocco di controllo. Di conseguenza, l'uso di funzioni `make` per creare oggetti di un tipo di classe che ha versioni specifiche di `operator new` e `operator delete` è generalmente una cattiva idea.

I vantaggi dimensionali e prestazionali di `std::make_shared` rispetto all'uso diretto di `new` derivano dal fatto che il blocco di controllo di `std::shared_ptr` viene inserito nello stesso frammento di memoria dell'oggetto gestito. Quando il conteggio dei riferimenti dell'oggetto va a 0, l'oggetto viene distrutto (ovvero viene richiamato il suo distruttore). Tuttavia, la memoria che occupa non può essere rilasciata finché non è stato distrutto anche il blocco di controllo, poiché entrambi sono contenuti nello stesso frammento di memoria, allocato dinamicamente.

Come ho detto, il blocco di controllo contiene informazioni gestionali che vanno oltre il semplice conteggio dei riferimenti. Il conteggio dei riferimenti registra quanti `std::shared_ptr` fanno riferimento al blocco di controllo, ma il blocco di controllo contiene un secondo conteggio dei riferimenti, che conta anche quanti `std::weak_ptr` fanno riferimento al blocco di controllo. Questo secondo conteggio dei riferimenti è noto come *conteggio debole*.[10] Quando uno `std::weak_ptr` controlla se è scaduto (vedi Elemento 19), lo fa esaminando il conteggio dei riferimenti (non il conteggio debole) nel blocco di controllo cui fa riferimento. Se il conteggio dei riferimenti è arrivato a 0 (ovvero, se l'oggetto puntato non ha più alcuno `std::shared_ptr` che vi fa riferimento ed è pertanto stato distrutto), lo `std::weak_ptr` è scaduto. Altrimenti no.

Fintantoché gli `std::weak_ptr` fanno riferimento al blocco di controllo (ovvero, il conteggio debole è maggiore di 0), tale blocco di controllo deve continuare a esistere. E fintantoché esiste un blocco di controllo, la memoria che lo contiene deve rimanere allocata. La memoria allocata dalla funzione `make` è di uno `std::shared_ptr`, e pertanto non può essere deallocata finché non sono stati

distrutti l'ultimo `std::shared_ptr` *e* anche l'ultimo `std::shared_ptr` che vi fa riferimento.

Se il tipo dell'oggetto è piuttosto esteso e il tempo fra la distruzione dell'ultimo std::shared_ptr e l'ultimo `std::weak_ptr` è significativo, può verificarsi un ritardo fra il momento in cui l'oggetto viene distrutto e il momento in cui la memoria che occupava viene liberata:

```
class ReallyBigType { … };

auto pBigObj =                    // crea un oggetto
  std::make_shared<ReallyBigType> // molto grosso
  ();
                                  // tramite std::make_shared


…                   // crea degli std::shared_ptr e std::weak_ptr
                    verso il
                    // grosso oggetto, e li usa per elaborarlo

…                   // distruzione dell'ultimo std::shared_ptr
                    all'oggetto,
                    // ma permanenza degli std::weak_ptr

…                   // Durante questo periodo, la memoria
                    precedentemente
                    // occupata dal grosso oggetto rimane allocata

…                   // distruzione dell'ultimo std::weak_ptr
                    all'oggetto;
                    // solo ora viene rilasciata la memoria per il
                    blocco
                    // di controllo E l'oggetto
```

Con l'utilizzo diretto di `new`, la memoria per l'oggetto `ReallyBigType` può essere rilasciata non appena è stato distrutto l'ultimo `std::shared_ptr` che vi punta:

```
class ReallyBigType // come prima
{ ... };
```

```
std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                    // crea un grosso oggetto
                    // tramite new
…                   // come prima, crea degli std::shared_ptr e
                    // std::weak_ptr all’oggetto, e li usa per
                    elaborarlo
                    // distruzione dell’ultimo std::shared_ptr
…                   all’oggetto,
                    // ma permanenza degli std::weak_ptr
                    // Durante questo periodo la memoria per
                    l’oggetto
…                   // viene deallocata, mentre rimane allocata
                    // solo la memoria per il blocco di controllo
                    // distruzione dell’ultimo std::weak_ptr
…                   all’oggetto;
                    // ora viene rilasciata la memoria per il
                    blocco di controllo
```

Se vi doveste trovare in una situazione in cui l’uso di `std::make_shared` è impossibile o inappropriato, dovreste fare attenzione ai problemi di sicurezza nella gestione delle eccezioni cui abbiamo accennato in precedenza. Il miglior modo per farlo è assicurarsi che quando si usa direttamente `new`, si passi immediatamente il risultato al costruttore di un puntatore smart in un’istruzione che fa questo e solo questo. Ciò evita ai compilatori di generare codice che potrebbe emettere un’eccezione fra l’uso del `new` e la chiamata del costruttore del puntatore smart che gestirà l’oggetto creato con `new`.

Come esempio, considerate una piccola revisione della chiamata “non sicura” in termini di gestione delle eccezioni alla funzione `processWidget` che abbiamo esaminato in precedenza. Questa volta, specificheremo un cancellatore custom:

```
void
processWidget(std::shared_ptr<Widget>   // come prima
spw.
    int priority);
void cusDel(Widget *ptr);               // cancellatore
                                        // custom
```

Ecco la chiamata non sicura alle eccezioni:

```
processWidget(                                   // come prima,
   std::shared_ptr<Widget>(new Widget,           // potenziale
   cusDel),
   computePriority()                             // spreco
);                                               // di risorse!
```

Ricapitoliamo: se `computePriority` viene richiamata dopo `new Widget`, ma prima del costruttore di `std::shared_ptr` e se `computePriority` lancia un'eccezione, il `Widget` allocato dinamicamente produrrà uno spreco di memoria.

Qui l'uso di un cancellatore custom preclude l'uso di `std::make_shared`, pertanto il modo per evitare il problema consiste nell'inserire l'allocazione del `Widget` e la costruzione dello `std::shared_ptr` in una propria istruzione e poi richiamare `processWidget` con lo `std::shared_ptr` risultante. Questa è la base della tecnica, ma come vedremo fra breve, possiamo modificarla per migliorarne le prestazioni:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority()); // corretto, ma non
                                           // ottimale; vedi sotto
```

Questo funziona, poiché uno `std::shared_ptr` assume la proprietà del puntatore standard passato al suo costruttore, anche se tale costruttore lancia un'eccezione. In questo esempio, se il costruttore di `spw` lancia un'eccezione (per esempio a causa di un'impossibilità di allocare dinamicamente la memoria per un blocco di controllo), è comunque garantito che sul puntatore risultante da `new Widget` verrà richiamata `cusDel`.

Vi è una piccola inefficienza legata al fatto che in una chiamata non sicura alle eccezioni, passiamo a `processWidget` un rvalue,

```
processWidget(
  std::shared_ptr<Widget>(new Widget,            // arg è un rvalue
  cusDel),
  computePriority()
);
```

mentre nella chiamata sicura passiamo un lvalue:

```
processWidget(spw, computePriority());   // arg è un lvalue
```

Poiché il parametro `std::shared_ptr` di `processWidget` viene passato per valore, la costruzione da un rvalue prevede solo uno spostamento, mentre la costruzione da un lvalue richiede una copia. Per uno `std::shared_ptr`, la differenza può essere significativa, poiché la copia di uno `std::shared_ptr` richiede un incremento atomico del suo conteggio dei riferimenti, mentre lo spostamento di uno `std::shared_ptr` non richiede alcuna manipolazione del conteggio dei riferimenti. Perché il codice sicuro alle eccezioni possa raggiungere un livello prestazionale pari a quello del codice non sicuro, occorre applicare `std::move` a `spw`, per trasformarlo in un rvalue (vedi Elemento 23):

```
processWidget(std::move(spw), // efficiente e
      computePriority());    // sicuro alle eccezioni
```

Questo è un aspetto interessante e vale la pena di memorizzarlo, ma normalmente è anche irrilevante, poiché raramente vi è un motivo per non utilizzare una funzione `make`. E a meno che abbiate un bisogno assoluto di fare diversamente, dovreste comunque impiegare una funzione `make`.

Argomenti da ricordare

- Rispetto all'uso diretto di `new`, le funzioni `make` eliminano la duplicazione del codice sorgente, migliorano la sicurezza nella gestione delle eccezioni e, per `std::make_shared` e `std::allocate_shared`, generano codice più compatto e veloce.
- Fra le situazioni in cui l'uso di funzioni `make` è inappropriato, vi è la necessità di specificare cancellatori custom e l'intenzione di passare inizializzatori a graffe.
- Per gli `std::shared_ptr`, ulteriori situazioni in cui le funzioni `make` sono sconsigliabili comprendono (1) le classi con gestione personalizzata della

memoria e (2) i sistemi che hanno vincoli di memoria, gli oggetti di grandi dimensioni e gli `std::weak_ptr` che sopravvivono ai rispettivi `std::shared_ptr`.

Elemento 22 – Idioma Pimpl: definire speciali funzioni membro nel file di implementazione

Se avete mai dovuto combattere contro eccessivi tempi di compilazione, conoscerete l'*idioma Pimpl* ("Pointer to IMPLementation"). Si tratta della tecnica mediante la quale si sostituiscono i dati membro di una classe con un puntatore a una classe di implementazione (o struttura), si inseriscono nella classe di implementazione i dati membro che originariamente si trovavano nella classe primaria e si accede a tali dati membri in modo indiretto tramite il puntatore. Per esempio, immaginate che `Widget` abbia il seguente aspetto:

```
class Widget {          // nell'header widget.h
public:
   Widget();
   …
private:
   std::string name;
   std::vector<double>
   data;
   Gadget g1, g2, g3;  // Gadget è un tipo
};                      // definito dall'utente
```

Poiché i dati-membro di `Widget` sono di tipo `std::string`, `std::vector` e `Gadget`, perché `Widget` possa essere compilato, gli header di questi tipi devono essere presenti e ciò significa che i clienti di `Widget` devono impiegare `#include <string>`, `<vector>` e `gadget.h`. Questi header allungano i tempi di compilazione per i client di `Widget;` inoltre rendono tali client dipendenti dal contenuto degli header. Se il contenuto di un header dovesse cambiare, i client di Widget dovranno essere ricompilati. Gli header standard `<string>` e `<vector>` non cambiano molto spesso, ma è probabile che `gadget.h` sia soggetto a frequenti revisioni.

Applicando l’idioma Pimpl in C++98, non si potrebbero sostituire i dati membro di `Widget` con un puntatore standard verso una struttura che viene dichiarata, ma non definita:

```
class Widget {        // sempre nell’header widget.h
public:
   Widget();
   ~Widget();         // il distr. è necessario, vedi sotto
…
private:
   struct Impl;       // dichiara la struct dell’implementazione
   Impl *pImpl;       // e il relativo puntatore
};
```

Poiché `Widget` non menziona più i tipi `std::string`, `std::vector` e `Gadget`, i client di `Widget` non hanno più bisogno di includere gli header per questi tipi. Ciò accelera la compilazione e significa anche che se qualcosa in questi header dovesse cambiare, i client di `Widget` non subirebbero conseguenze.

Un tipo che è stato dichiarato, ma non definito, è un *tipo incompleto*. `Widget::Impl` è proprio questo genere di tipo. Vi sono poche cose che si possono fare su un tipo incompleto, ma una di queste è la dichiarazione di un puntatore. L’idioma Pimpl sfrutta proprio questa possibilità.

La prima parte dell’idioma Pimpl è la dichiarazione di un dato membro che è un puntatore a un tipo incompleto. La seconda parte è l’allocazione e deallocazione dinamica dell’oggetto che contiene i dati membro che originariamente si trovavano nella classe. Il codice di allocazione e deallocazione va nel file di implementazione, per esempio `widget.cpp`, per Widget:

```
#include “widget.h”              // nel file di impl. widget.cpp
#include “gadget.h”
#include <string>
#include <vector>

struct Widget::Impl {            // definizione di Widget::Impl
   std::string name;             // con i dati-membro
   std::vector<double> data;     // precedentemente in Widget
```

```
   Gadget g1, g2, g3;
};

Widget::Widget()                    // alloca i dati-membro
: pImpl(new Impl)                   // per questo oggetto Widget
{}

Widget::~Widget()                   // distrugge i dati-membro
{ delete pImpl; }                   // per questo oggetto
```

Rimettiamo le direttive `include` per chiarire il fatto che le dipendenze dagli header per `std::string`, `std::vector` e `Gadget` continuano a sussistere. Tuttavia, queste dipendenze sono state spostate da `widget.h` (che risulta visibile e utilizzabile da parte dei client di `Widget`) a `widget.cpp` (che è visibile e utilizzabile solo dall'implementatore di `Widget`). Ho anche evidenziato il codice che alloca e dealloca dinamicamente l'oggetto di tipo `Impl`. La necessità di deallocare questo oggetto quando viene distrutto un `Widget`, richiede la presenza di un distruttore per `Widget`.

Ma si tratta di codice C++98, del millennio scorso. Utilizza solo puntatori standard, solo `new` standard e solo `delete` standard e pertanto è tutto così grezzo! Questo capitolo si basa sull'idea che i puntatori smart siano preferibili ai puntatori standard e, se quello che vogliamo fare è allocare dinamicamente un oggetto `Widget::Impl` all'interno del costruttore di `Widget` e far sì che venga distrutto contemporaneamente a `Widget`, lo strumento di cui abbiamo bisogno è `std::unique_ptr` (vedi Elemento 18). Sostituendo il puntatore standard `pImpl` con uno `std::unique_ptr` si ottiene il seguente codice per il file header,

```
class Widget {                             // in widget.h
public:
   Widget();
   …

private:
struct Impl;
   std::unique_ptr<Impl> pImpl;            // usa un puntatore smart
};                                         // invece di un puntatore
```

```
                                        standard
```

e il seguente file di implementazione:

```
#include "widget.h"                     // in widget.cpp
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {                   // come prima
   std::string name;
   std::vector<double> data;
   Gadget g1, g2, g3;
};

Widget::Widget()                        // vedi Elemento 21: crea
: pImpl(std::make_unique<Impl>())       // std::unique_ptr
{}                                      // tramite
                                        std::make_unique
```

Noterete che il distruttore di `Widget` non è più presente. Questo perché non è più necessario inserirvi del codice. `std::unique_ptr` cancella automaticamente ciò a cui punta, quando viene distrutto (lo `std::unique_ptr`), quindi non dobbiamo più preoccuparci di cancellare nulla. Questo è uno dei vantaggi dei puntatori smart: eliminano la necessità di occuparsi manualmente del rilascio delle risorse.

Questo codice può essere compilato, ma, purtroppo, il suo utilizzo più semplice all'interno dei client non passa la compilazione:

```
#include "widget.h"
Widget w;           // errore!
```

Il messaggio d'errore ricevuto dipende dal compilatore utilizzato, ma il testo, generalmente, dirà qualcosa sul fatto che abbiamo applicato `sizeof` o `delete` a un tipo incompleto. Queste non sono contemplate fra le operazioni che potete fare con questi tipi.

Questo apparente fallimento dell'idioma Pimpl nell'utilizzo di `std::unique_ptr`

è allarmante, poiché (1) in teoria `std::unique_ptr` dovrebbe supportare i tipi incompleti e (2) l'idioma Pimpl è uno dei più comuni casi d'uso di `std::unique_ptr`. Fortunatamente, è piuttosto facile ottenere codice funzionante. Basta risalire alla causa del problema.

Il problema sorge a causa del codice che viene generato quando deve essere distrutta `w` (ovvero quando esce dal campo di visibilità, *scope*). A quel punto, viene richiamato il suo distruttore. Nella definizione della classe che utilizza lo `std::unique_ptr`, non dichiariamo un distruttore, poiché non abbiamo del codice da inserirvi. In base alle regole consuete per le funzioni membro speciali generate dal compilatore (vedi Elemento 17), il compilatore genera automaticamente un distruttore. All'interno di tale distruttore, il compilatore inserisce del codice per richiamare il distruttore per il dato membro di `Widget`, `pImpl`. `pImpl` è uno `std::unique_ptr<Widget::Impl>`, ovvero uno `std::unique_ptr` che utilizza il cancellatore di default. Il cancellatore di default è una funzione che applica `delete` al puntatore standard che si trova all'interno dello `std::unique_ptr`. Prima di utilizzare `delete`, tuttavia, in genere le implementazioni fanno sì che il cancellatore di default impieghi una `static_assert` del C++11 per garantire che il puntatore standard non punti a un tipo incompleto. Quando il compilatore genera il codice per la distruzione del `Widget w`, quindi, generalmente incontra una `static_assert` che fallisce e questo è ciò che causa il messaggio d'errore. Questo messaggio è associato al punto in cui `w` viene distrutta, poiché il distruttore di `Widget`, come tutte le funzioni membro speciali generate dal compilatore, è implicitamente `inline`. Il messaggio stesso, spesso fa riferimento alla riga in cui viene creata `w`, poiché è il codice sorgente che crea esplicitamente l'oggetto che provoca, successivamente, la sua distruzione implicita.

Per correggere il problema, basta assicurarsi che nel punto in cui viene generato il codice per distruggere lo `std::unique_ptr<Widget::Impl>`, `Widget::Impl` sia un tipo completo. Il tipo diviene completo quando si trova la sua definizione e `Widget::Impl` è definito all'interno di `widget.cpp`. Perché la compilazione abbia successo, quindi, il compilatore deve vedere il corpo del distruttore di `Widget` (ovvero, il luogo in cui il compilatore genererà il codice per distruggere il dato membro `std::unique_ptr`) solo all'interno del file `widget.cpp`, dopo che `Widget::Impl` è stato definito.

La soluzione è semplice. Basta dichiarare il distruttore del `Widget` in `widget.h`, ma non definirlo:

```
class Widget {           // come prima, in widget.h
public:
   Widget();
   ~Widget();            // sola dichiarazione
   …

private:                 // come prima
   struct Impl;
   std::unique_ptr<Impl>
   pImpl;
};
```

Lo si deve definire in `widget.cpp`, dopo che è stato definito Widget::Impl:

```
#include “widget.h”                       // come prima, in
                                          widget.cpp
#include “gadget.h”
#include <string>
#include <vector>

   struct Widget::Impl {                  // come prima, definizione
                                          di
   std::string name;                      // Widget::Impl
   std::vector<double> data;
   Gadget g1, g2, g3;
};

Widget::Widget()                          // come prima
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()                         // definizione di ~Widget
{}
```

Questa soluzione funziona e richiede anche poco impegno di digitazione, ma se volete enfatizzare il fatto che il distruttore generato dal compilatore svolgerebbe

l'operazione corretta (e l'unico motivo per cui lo avevamo dichiarato era di far sì che la sua definizione venisse generata nel file di implementazione di `Widget`) potete definire il corpo del distruttore con `= default`:

```
Widget::~Widget() = default; // stesso effetto, come sopra
```

Le classi che utilizzano l'idioma Pimpl sono candidate naturali per il supporto dello spostamento, poiché le operazioni di spostamento generate dal compilatore fanno esattamente ciò che si desidera: uno spostamento sullo `std::unique_ptr` sottostante. Come spiega l'Elemento 17, la dichiarazione di un distruttore in `Widget` impedisce ai compilatori di generare le operazioni di spostamento. Pertanto se si vuole ottenere il supporto per lo spostamento, occorre dichiarare le funzioni esplicitamente. Dato che le versioni generate dal compilatore funzionerebbero correttamente, sareste tentati di implementarle nel seguente modo:

```
class Widget {        // sempre in
public:               // widget.h
   Widget();
   ~Widget();

   Widget(Widget&& rhs) = default;                  // buona idea,
   Widget& operator=(Widget&& rhs) = default;       // codice sbagliato!

…

private:                                             // come prima
   struct Impl;
   std::unique_ptr<Impl> pImpl;
};
```

Questo approccio porta allo stesso tipo di problema che si ha dichiarando la classe senza un distruttore e, sostanzialmente, per lo stesso motivo. L'operatore di assegnamento per spostamento generato dal compilatore deve distruggere l'oggetto puntato da `pImpl` prima di riassegnarlo, ma nel file header di `Widget`, `pImpl` punta a un tipo incompleto. La situazione è differente per il costruttore per spostamento. Il problema qui è che i compilatori, generalmente, generano del

codice per distruggere `pImpl` nel caso in cui venga lanciata un'eccezione all'interno del costruttore per spostamento e la distruzione di `pImpl` richiede che `pImpl` sia ormai completo.

Poiché il problema è lo stesso già visto prima, anche la soluzione lo è: spostare la definizione delle operazioni di spostamento all'interno del file di implementazione:

```
class Widget {                              // sempre in widget.h
public:
   Widget();
   ~Widget();

   Widget(Widget&& rhs);                    // solo
   Widget& operator=(Widget&& rhs);         // dichiarazioni

   …

private:                                    // come prima
   struct Impl;
   std::unique_ptr<Impl> pImpl;
};
#include <string>                           // come prima,
…                                           // in "widget.cpp"

struct Widget::Impl { ... };                // come prima

Widget::Widget()                            // come prima
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;                // come prima

Widget::Widget(Widget&& rhs) = default; // definizioni
Widget& Widget::operator=(Widget&& rhs) = default;
```

L'idioma Pimpl è un modo per ridurre le dipendenze in fase di compilazione fra

l’implementazione di una classe e i client della classe, ma, sostanzialmente, l’uso di questo idioma non cambia ciò che la classe rappresenta. La classe `Widget` originale conteneva i dati membro `std::string`, `std::vector` e `Gadget`; supponendo che i `Gadget`, le `std::string` e gli `std::vector` possano essere copiati, sarebbe sensato che `Widget` supportasse le operazioni di copia. Dobbiamo riscrivere queste funzioni esplicitamente, poiché (1) i compilatori non generano le operazioni di copia per le classi con tipi move-only come `std::unique_ptr` e (2) anche se lo facessero, le funzioni generate copierebbero solo lo `std::unique_ptr` (ovvero eseguirebbero una *copia superficiale*) mentre vogliamo copiare ciò cui punta il puntatore (ovvero eseguire una *copia profonda*).

Secondo un rituale che, a questo punto, dovrebbe essere familiare, dichiariamo le funzioni nel file header e le implementiamo nel file di implementazione:

```
class Widget {                              // sempre in widget.h
public:
…                                           // altre funzioni, come prima

   Widget(const Widget& rhs);               // solo
   Widget& operator=(const Widget& rhs);    // dichiarazioni

private:                                    // come prima
   struct Impl;
   std::unique_ptr<Impl> pImpl;
};

#include “widget.h”                         // come prima,
…                                           // in “widget.cpp”

struct Widget::Impl { ... };                // come prima

Widget::~Widget() = default;                // altre funzioni, come prima

Widget::Widget(const Widget& rhs)           // costr. per copia
```

```
: pImpl(std::make_unique<Impl>
(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget&     // operatore= per copia
rhs)
{
   pImpl = rhs.pImpl;
   return *this;
}
```

L'implementazione di entrambe le funzioni è convenzionale. In entrambi i casi, copiamo semplicemente i campi della struttura `Impl` dall'oggetto di origine (`rsh`) all'oggetto di destinazione (`*this`). Invece di copiare i campi uno per uno, sfruttiamo il fatto che i compilatori creeranno le operazioni di copia per `Impl` e queste operazioni copieranno automaticamente ciascun campo. Pertanto, implementiamo le operazioni di copia di `Widget` richiamando le operazioni di copia di `Widget::Impl` generate dal compilatore. Nel costruttore per copia, notate che seguiamo comunque il consiglio fornito nell'Elemento 21 di preferire `std::make_unique` all'utilizzo diretto di `new`.

Per l'implementazione dell'idioma Pimpl, `std::unique_ptr` è il puntatore smart da utilizzare, poiché il puntatore `pImpl` all'interno di un oggetto (ovvero all'interno di un `Widget`) ha la proprietà esclusiva dell'oggetto di implementazione corrispondente (ovvero l'oggetto `Widget::Impl`). In ogni caso, è interessante notare che se per `pImpl` dovessimo utilizzare `std::shared_ptr` al posto di `std::unique_ptr`, scopriremmo che il consiglio fornito per questo punto non si applica più. Non vi sarebbe più alcuna necessità di dichiarare un distruttore in `Widget`. E senza un distruttore dichiarato dall'utente, i compilatori genereranno tranquillamente le operazioni di spostamento, le quali faranno esattamente ciò che devono. Per questo motivo, dato il seguente codice in `widget.h`,

```
class Widget {               // in widget.h
public:
   Widget();
   …                         // niente dichiarazioni per le
                             operazioni
```

```
                              // di distruzione e spostamento
private:
   struct Impl;
   std::shared_ptr<Impl>
   pImpl;                     // std::shared_ptr
};
```

e il seguente codice client #include widget.h,

```
Widget w1;

auto w2(std::move(w1));     // costruisce per spostamento w2



w1 = std::move(w2);         // assegna per spostamento w1
```

tutto verrebbe compilato ed eseguito correttamente: `w1` verrebbe costruito tramite il default, il suo valore verrebbe spostato in `w2`, tale valore verrebbe spostato nuovamente in `w1` e poi sia `w1` sia `w2` verrebbero distrutti (provocando pertanto la distruzione dell'oggetto puntato `Widget::Impl`).

La differenza di comportamento fra `std::unique_ptr` e `std::shared_ptr` per i puntatori `pImpl` deriva dai modi differenti con cui questi puntatori smart supportano i cancellatori custom. Per `std::unique_ptr`, il tipo del cancellatore fa parte del tipo del puntatore smart e ciò consente ai compilatori di generare strutture dati runtime più compatte e codice runtime più veloce. Una conseguenza di questa maggiore efficienza è il fatto che i tipi puntati devono essere completi quando vengono utilizzate le funzioni speciali generate dal compilatore (per esempio i distruttori o le operazioni di spostamento). Per `std::shared_ptr`, il tipo del cancellatore non fa parte del tipo del puntatore smart. Ciò richiede strutture dati runtime di maggiori dimensioni e codice più lento, ma i tipi puntati non devono necessariamente essere completi quando vengono impiegate le funzioni speciali generate dal compilatore.

Per l'idioma Pimpl, non esiste davvero un compromesso fra le caratteristiche di `std::unique_ptr` e quelle di `std::shared_ptr`, poiché la relazione fra le classi come `Widget` e le classi come `Widget::Impl` è di proprietà esclusiva e ciò rende `std` ::unique_ptr lo strumento perfetto per questo compito. Ciononostante, vale la pena di sapere che in altre situazioni, in cui si applica la proprietà condivisa (e

pertanto std::shared_ptr è una scelta progettuale appropriata), non vi è alcuna necessità di ricorrere alle tecniche di definizione delle funzioni previste da `std::unique_ptr`.

Argomenti da ricordare

- L'idioma Pimpl abbatte i tempi di compilazione, riducendo le dipendenze di compilazione fra i client della classe e l'implementazione della classe.
- Per i puntatori `std::unique_ptr pImpl`, occorre dichiarare le funzioni membro speciali nell'header della classe, ma implementarle nel file di implementazione. Lo si deve fare anche se le implementazioni di default della funzione sono accettabili.
- Il consiglio precedente si applica a `std::unique_ptr`, ma non a `std::shared_ptr`.

---

7. Vi sono alcune eccezioni a questa regola. La maggior parte di esse deriva dalla chiusura anormale del programma. Se un'eccezione si propaga fino al livello della funzione principale del thread (per esempio `main`, per il thread iniziale del programma) o se viene violata una specifica `noexcept` (vedi Elemento 14), gli oggetti locali potrebbero non essere distrutti; e se viene richiamata `std::abort` o una funzione di uscita (per esempio `std::_Exit`, `std::exit` o `std::quick_exit`), decisamente non verranno distrutti.

8. Questa implementazione non è richiesta dallo Standard, ma ogni implementazione nota della Libreria Standard la impiega.

9. Per creare una `make_unique` ricca di funzionalità con la minore fatica possibile, ricercate il documento di standardizzazione che ha dato origine al tutto, poi copiate l'implementazione che vi trovate. Il documento che cercate è il N3656 di Stephan T. Lavavej, del 18 aprile 2013.

10. In realtà, il valore del conteggio debole non è sempre uguale al numero di `std::weak_ptr` che fanno riferimento al blocco di controllo, poiché gli implementatori di librerie hanno trovato dei modi per inserire ulteriori informazioni nel conteggio debole, con lo scopo di migliorare la generazione del codice. Per gli scopi di questo Elemento, ignoreremo questo fatto e supporremo

che il valore del conteggio debole sia equivalente al numero di `std::weak_ptr` che fanno riferimento al blocco di controllo.

5

Riferimenti rvalue, semantica di spostamento e perfect-forward

Da principio, la semantica dello spostamento e il perfect-forward sembrano concetti piuttosto semplici.

- **La semantica di spostamento** consente ai compilatori di sostituire delle costose operazioni di copia con meno costosi spostamenti. Nello stesso modo in cui i costruttori per copia e gli operatori di assegnamento per copia offrono il controllo sulla copia degli oggetti, i costruttori per spostamento e gli operatori d'assegnamento per spostamento offrono il controllo sulla semantica dello spostamento. La semantica della spostamento consente inoltre di creare tipi move-only, come `std::unique_ptr`, `std::future` e `std::thread`.
- **Il perfect-forward** consente di scrivere modelli di funzioni (template) che accettano argomenti arbitrari e li inoltrano ad altre funzioni, in modo che le funzioni di destinazione ricevano esattamente gli stessi argomenti passati alle funzioni che eseguono l'inoltro.

I riferimenti rvalue sono il collante che unisce queste due funzionalità. Sono il meccanismo offerto dal linguaggio per rendere possibile sia la semantica dello spostamento sia il perfect-forward.

Ma più esperienza si acquisisce su queste funzionalità, più si comprende che l'impressione iniziale si basava solo sulla proverbiale "punta dell'iceberg". Il mondo della semantica di spostamento, del perfect-forward e dei riferimenti rvalue è più intricato di quanto sembri. Per esempio, `std::move` non sposta nulla, e il perfect-forward è tutt'altro che "perfetto". Le operazioni di spostamento non sempre sono più economiche rispetto a quelle di copia; quando lo sono, non sono tanto economiche quanto si potrebbe immaginare; e non sempre vengono richiamate in un contesto in cui lo spostamento è valido. Il costrutto "***type*&&**" non sempre rappresenta un riferimento rvalue.

Indipendentemente da quanto si approfondiscano queste funzionalità, c'è sempre l'impressione che ci sia ancora qualcosa da scoprire. Fortunatamente, esiste un limite a questa profondità. Questo capitolo vi accompagnerà proprio fin sul fondo. Una volta che vi arriverete, questo aspetto del C++11 vi sarà molto più chiaro. Per esempio, conoscerete le convenzioni d'uso per `std::move` e `std::forward`. Vi troverete a vostro agio con la natura ambigua di "***type*&&**". Comprenderete i motivi, talvolta sorprendenti, degli strani comportamenti delle operazioni di spostamento. Tutti questi dettagli avranno una loro precisa collocazione. A quel punto, in modo paradossale, vi ritroverete dove siete partiti, poiché la semantica dello spostamento, il perfect-forward e i riferimenti rvalue torneranno a sembrarvi concetti semplici. Ma, a quel punto, li avrete davvero compresi appieno.

Negli Elementi di questo Capitolo, è particolarmente importante tenere in considerazione che un parametro è sempre un lvalue, anche se il suo tipo è un riferimento rvalue. Detto questo, dato

```
void f(Widget&& w);
```

il parametro `w` è un lvalue, anche se il suo tipo è un riferimento rvalue a `Widget` (se lo trovate sorprendente, rileggete la panoramica su `lvalues` e `rvalues` presente nell'Introduzione, all'interno del paragrafo Terminologia e convenzioni).

Elemento 23 – Parliamo di std::move e std::forward

È utile iniziare a parlare di `std::move` e `std::forward` in termini di ciò che *non*

fanno. `std::move` non sposta niente. `std::forward` non inoltra niente. In fase di esecuzione del programma, nessuno dei due fa nulla. Non generano codice eseguibile, neppure un unico byte.

`std::move` e `std::forward` sono semplicemente funzioni (in realtà template di funzioni) che eseguono conversioni. `std::move` converte incondizionatamente il proprio argomento in un rvalue, mentre std::forward svolge questa conversione solo se è esaudita una determinata condizione. Questo è tutto. Naturalmente una descrizione così sintetica fa sorgere nuove domande, ma, in buona sostanza, questo è tutto.

Per rendere più concreto il tutto, ecco un esempio di implementazione di std::move in C++11. Non è pienamente conforme ai dettagli dello Standard, ma gli si avvicina molto.

```
template<typename T>                        // nel namespace std
typename remove_reference<T>::type&&
move(T&& param)
{
   using ReturnType =                       // dichiarazione alias;
    typename
    remove_reference<T>::type&&;            // vedi Elemento 9

   return static_cast<ReturnType>
   (param);
}
```

Ho evidenziato due parti di codice. Un oè il nome della funzione, poiché la specifica del tipo restituito è piuttosto "rumorosa" e non voglio che in questo rumore possiate perdere dei dettagli. L'altra è la conversione che costituisce l'essenza della funzione. Come potete vedere, std::move prende un riferimento a un oggetto (un riferimento universale, per essere precisi, vedi Elemento 24) e restituisce un riferimento allo stesso oggetto.

La parte `&&` del tipo restituito dalla funzione implica che `std::move` restituisce un riferimento rvalue, ma, come spiega l'Elemento 28, se il tipo `T` è un riferimento lvalue, `T&&` diverrebbe un riferimento lvalue. Per impedire ciò, a `T` viene applicato il tipe trait (vedi Elemento 9) `std::remove_reference`,

garantendo così che && venga applicato a un tipo che non sia un riferimento. Ciò garantisce che `std::move` restituisca davvero un riferimento rvalue, e questo è importante, poiché i riferimenti rvalue restituiti dalle funzioni sono rvalue. Pertanto, `std::move` converte il proprio argomento in un rvalue, e questo è tutto ciò che fa.

A proposito, std::move può essere implementata con molta più facilità in C++14. Grazie alla deduzione del tipo restituito dalle funzioni (vedi Elemento 3) e al template alias `std::remove_reference_t` della Libreria Standard (vedi Elemento 9), std::move può essere scritta in questo modo:

```
template<typename T>                        // C++14; sempre nel
decltype(auto) move(T&& param)              // namespace std
{
   using ReturnType =
   remove_reference_t<T>&&;
   return static_cast<ReturnType>
   (param);
}
```

Molto più leggibile, vero?

Poiché std::move non fa altro che convertire il proprio argomento in un rvalue, qualcuno ha suggerito che un nome migliore per questa funzione sarebbe stato qualcosa come `rvalue_cast`. Sia come sia, il nome utilizzato è std::move, pertanto è importante ricordare ciò che `std::move` fa e non fa. Esegue una conversione, non uno spostamento.

Naturalmente, gli rvalue sono candidati per gli spostamenti, dunque l'applicazione di `std::move` a un oggetto comunica al compilatore che l'oggetto può essere spostato. Questo è il motivo per cui `std::move` ha questo nome: per designare gli oggetti che possono essere spostati.

In verità, gli rvalue "solitamente" sono candidati per lo spostamento. Supponete di dover scrivere una classe che rappresenta generiche annotazioni. Il costruttore della classe prende un parametro `std::string` che costituisce l'annotazione e copia il parametro in un dato membro. Come descritto nell'Elemento 41, si dichiara un parametro per valore:

```
class Annotation {
public:
   explicit Annotation(std::string text);   // parametro da copiare,
   …                                        // vedi Elemento 41,
};                                          // passaggio per valore
```

Ma il costruttore di `Annotation` ha bisogno solo di leggere il valore di `text`. Non deve modificarlo. Seguendo la lunga tradizione di utilizzare `const` quando possibile, si modifica la dichiarazione in modo che `text` sia `const`:

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  …
};
```

Per evitare di pagare per un'operazione di copia quando si deve copiare `text` in un dato membro, ci atteniamo al consiglio fornito nel Punto 41 e applichiamo `std::move` a `text`, producendo pertanto un rvalue:

```
class Annotation {
public:
   explicit Annotation(const std::string text)
   : value(std::move(text))     // “move” di text in value; questo codice
   { ... }                      // non fa quello che sembra!

   …

private:
   std::string value;
};
```

Questo codice passa la compilazione. Passa anche il linking. Viene anche seguito. Questo codice assegna al dato membro `value` il contenuto di `text`. L'unica cosa che separa questo codice da una realizzazione perfetta di questa

idea è che `text` non viene spostato in `value`, ma viene *copiato*. Certamente, `text` viene convertito in un rvalue da `std::move`, ma `text` è dichiarato come una `const std::string`, pertanto prima della conversione, `text` è un lvalue const `std::string` e il risultato della conversione è un rvalue const `std::string`, ma, attraverso tutto questo, rimane il fatto che `text` è `const`.

Considerate l'effetto che si ha quando i compilatori devono determinare quale costruttore di `std::string` richiamare. Hanno due possibilità:

```
class string {                          // std::string è in
                                        effetti un
public:                                 // typedef per std::basic
                                        string<char>
    …
    string(const string& rhs);          // costr. per copia
    string(string&& rhs);               // costr. per spostamento
…
};
```

Nell'elenco di inizializzazione dei membri del costruttore di `Annotation`, il risultato di `std::move(text)` è un rvalue di tipo `const std::string`. Tale rvalue non può essere passato al costruttore per spostamento di `std::string`, poiché il costruttore per spostamento prende un riferimento rvalue a una `std::string` *non*-`const`. L'r value può, tuttavia, essere passato al costruttore per copia, poiché a un rvalue `const` può associarsi a un riferimento lvalue a `const`. L'inizializzazione del membro, pertanto, richiama il costruttore *per copia* di `std::string`, anche se `text` è stato convertito in un rvalue! Tale comportamento è fondamentale per mantenere la correttezza in termini di `const`. Lo spostamento di un valore fuori da un oggetto, generalmente, è per modificare tale oggetto, e pertanto il linguaggio non dovrebbe mai permettere che gli oggetti `const` vengano passati a delle funzioni (come i costruttori per copia) che potrebbero modificarli.

Da questo esempio si possono trarre due lezioni. Innanzitutto, non si devono dichiarare oggetti `const` se poi si vuole avere la possibilità di spostarli. Le richieste di spostamento su oggetti `const` vengono, di fatto, trasformate in operazioni di copia. In secondo luogo, non solo `std::move` in realtà non sposta nulla, ma non garantisce neppure che l'oggetto che sta convertendo sia

spostabile. L’unica cosa davvero certa sul risultato dell’applicazione di `std::move` a un oggetto è il fatto che si ottiene un rvalue.

A `std::forward` si applica un ragionamento simile a quello per `std::move`, ma mentre `std::move` converte *incondizionatamente* il proprio argomento in un rvalue, `std ::forward` lo fa solo in alcuni casi. `std::forward` è una conversione *condizionale*. Per capire quando esegue e quando non esegue la conversione, bisogna considerare il modo in cui `std::forward` viene normalmente utilizzata. La situazione più comune è un template di funzione che accetta un parametro riferimento universale che deve essere passato a un’altra funzione:

```
void process(const Widget& lvalArg);     // elabora gli lvalue
void process(Widget&& rvalArg);          // elabora gli rvalue

template<typename T>                     // template che passa
void logAndProcess(T&& param)            // i parametri da
                                         elaborare
{
   auto now =                            // ora corrente
      std::chrono::system clock::now();

   makeLogEntry(“Calling ‘process’”, now);
   process(std::forward<T>(param));
}
```

Considerate due chiamate a `logAndProcess`: una con un lvalue e l’altra con un rvalue:

```
Widget w;

logAndProcess(w);                         // chiamata con lvalue
logAndProcess(std::move(w));              // chiamata con rvalue
```

All’interno di `logAndProcess`, il parametro `param` viene passato alla funzione `process`. Quest’ultima ha un overloading per lvalue e un altro per rvalue. Quando si richiama `logAndProcess` con un lvalue, ci aspettiamo naturalmente

che tale lvalue venga inoltrato a `process` come un lvalue e quando richiamiamo `logAndProcess` con un rvalue, ci aspettiamo che venga richiamato l'overloading di `process` per gli rvalue.

Ma `param`, come tutti i parametri di una funzione, è un lvalue. Ogni chiamata a `process` all'interno di `logAndProcess` vorrà pertanto richiamare l'overloading lvalue di `process`. Per evitare ciò, abbiamo bisogno di un meccanismo grazie al quale `param` possa essere convertito in un rvalue se (e solo se) l'argomento con cui è stato inizializzato `param` (l'argomento passato a `logAndProcess`) era un rvalue. Questo è esattamente ciò che fa `std::forward`. Questo è il motivo per cui `std::forward` è una conversione *condizionale:* converte in un rvalue solo se il suo argomento è stato inizializzato con un rvalue.

Potreste chiedervi come faccia `std::forward` a sapere se il suo argomento è stato inizializzato con un rvalue. Nel codice precedente, per esempio, come può `std::forward` sapere se `param` è stato inizializzato con un lvalue o con un rvalue? La risposta breve è che tale informazione viene codificata nel parametro `T` del template di `logAndProcess`. Il parametro viene passato a `std::forward`, che estrae l'informazione codificata. Per i dettagli dell'esatto funzionamento di questo meccanismo, consultate l'Elemento 28.

Dato che sia `std::move` sia `std::forward`, alla fine, non sono altro che conversioni, dove l'unica differenza consiste nel fatto che `std::move` esegue la conversione *sempre*, mentre `std::forward` la esegue solo *qualche volta*, potreste chiedervi se non si possa lasciar perdere `std::move` e utilizzare sempre `std::forward`, ovunque. Dal punto di vista puramente tecnico, la risposta è affermativa: `std::forward` può fare tutto e `std ::move` non è necessaria. Naturalmente nessuna delle due funzioni è davvero necessaria, poiché potremmo anche scrivere direttamente le conversioni, ma credo che siamo tutti d'accordo sul fatto che questo sarebbe... indesiderabile.

I punti di forza di `std::move` sono la comodità, il fatto che riduce l'insorgere di errori e la maggiore chiarezza. Considerate una classe in cui vogliamo determinare quante volte viene richiamato il costruttore per spostamento. Predisponiamo un semplice contatore `static` che viene incrementato durante la costruzione per spostamento. Supponendo che l'unico dato non statico della classe sia una `std::string`, ecco il modo convenzionale (ovvero utilizzando `std::move`) per implementare il costruttore per spostamento:

```
class Widget {
```

```
public:
  Widget(Widget&& rhs)
  : s(std::move(rhs.s))
  { ++moveCtorCalls; }
  …
private:
  static std::size_t moveCtorCalls;
  std::string s;
};
```

Per implementare lo stesso comportamento con `std::forward`, il codice avrebbe il seguente aspetto:

```
class Widget {
public:
   Widget(Widget&& rhs)                  // unconventional,
   : s(std::forward<std::string>(rhs.s)) // indesiderabile
   { ++moveCtorCalls; }                  // implementation
…

};
```

Notate innanzitutto che `std::move` richiede un solo argomento `(rhs.s)`, mentre `std ::forward` richiede sia un argomento funzione `(rhs.s)` sia un argomento di tipo template `(std::string)`. Poi notate che il tipo che passiamo a `std::forward` non dovrà essere un riferimento, poiché per convenzione la codifica dell'argomento passato deve essere un rvalue (vedi Elemento 28). Ciò significa che `std::move` richiede meno impegno di digitazione rispetto a `std::forward` e ci evita anche la necessità di passare un argomento di tipo che codifica il fatto che l'argomento che stiamo passando è un rvalue. Inoltre, elimina la possibilità del passaggio di un tipo errato (per esempio `std ::string&`, che risulterebbe dal fatto che il dato membro `s` venga costruito per copia anziché per spostamento).

Ancora più importante: l'uso di `std::move` veicola la richiesta di una conversione incondizionata in un rvalue, mentre l'uso di `std::forward` veicola la richiesta di una conversione in un rvalue solo per i riferimenti cui sono associati degli rvalue. Si tratta di due azioni molto differenti. La prima, tipicamente, prevede uno spostamento, mentre la seconda non fa altro che

passare (inoltrare) un oggetto a un'altra funzione in un modo che mantiene le sue caratteristiche originali: lvalue o rvalue. Poiché si tratta di azioni così differenti, è giusto che esistano due funzioni differenti (con nomi differenti).

Argomenti da ricordare

- **`std::move`** esegue una conversione incondizionata in una rvalue e di per se stessa non sposta nulla.
- **`std::forward`** converte il proprio argomento in un rvalue solo se a l'argomento è associato a un rvalue.
- Né `std::move` né `std::forward` svolgono alcuna operazione runtime.

Elemento 24 – Distinguere i riferimenti universali e riferimenti rvalue

Si dice che la verità renda liberi, ma, in alcuni casi, una bugia ben scelta può essere altrettanto liberatoria. Questo Elemento è una bugia di questo tipo. Ma, in termini software, al posto di "bugia" si usa un termine più elegante: "astrazione".

Per dichiarare un riferimento rvalue a un determinato tipo `T`, si scrive `T&&`. Sembra pertanto ragionevole presumere che se si trova `T&&` nel codice sorgente, si tratti di un riferimento rvalue.

Purtroppo le cose non sono così semplici:

```
void f(Widget&& param);                 // riferimento rvalue

WidgetSS var1 = Widget();               // riferimento rvalue

autoSS var2 = var1;                     // non è riferimento
                                        rvalue

template<typename T>
void f(std::vector<T>SS param);         // riferimento rvalue
```

```
template<typename T>
void f(TSS param);                        // non è riferimento
                                          rvalue
```

In realtà, `T&&` ha due significati differenti. Uno, naturalmente, è un riferimento rvalue. Tali riferimenti si comportano esattamente nel modo previsto: si associano solo a rvalue e il loro *motivo di esistere* è proprio quello di indentificare gli oggetti da cui possono essere eseguiti spostamenti.

L'altro significato di `T&&` è quello di riferimento *rvalue oppure lvalue*. Tali riferimenti, nel codice sorgente, hanno entrambi l'aspetto di riferimenti rvalue (ovvero `T&&)`, ma possono comportarsi come se fossero riferimenti lvalue (ovvero `T&)`. Questa doppia natura permette loro di associarsi a rvalue (come riferimenti rvalue) o anche a lvalue (come riferimenti lvalue). Inoltre, possono associarsi a oggetti `const` o non**`-const`**, a oggetti `volatile` o non**`-volatile`** e perfino a oggetti `const` e `volatile`. Praticamente si possono legare a *qualsiasi cosa*. Tali riferimenti assolutamente flessibili meritano un nome tutto loro. Possiamo chiamarli *riferimenti universali.*[11]

I riferimenti universali si hanno in due contesti. Il più comune è quello dei parametri di template di funzioni, come questo esempio basato sul codice precedente:

```
template<typename T>
void f(TSS param);         // param è un riferimento universale
```

Il secondo contesto è costituito dalle dichiarazioni `auto`, inclusa la seguente, sempre tratta dal codice precedente:

```
auto&& var2 = var1;          // var2 è un riferimento universale
```

Questi due contesti hanno in comune la presenza della *deduzione del tipo.* Nel template di `f`, il tipo di `param` viene dedotto e, nella dichiarazione di `var2`, il tipo di `var2` viene dedotto. Confrontate questo con i seguenti esempi (sempre basati sul codice precedente), dove manca la deduzione del tipo. Se vedete `T&&` senza la deduzione del tipo, state osservando un riferimento rvalue:

```
void f(WidqetSS param);          // nessuna deduzione del tipo;
```

```
                                   // param è un riferimento rvalue

WidgetSS vari = Widqet();          // nessuna deduzione del tipo;
                                   // vari è un riferimento rvalue
```

Poiché i riferimenti universali sono “riferimenti”, devono essere inizializzati. È proprio l’inizializzatore di un riferimento universale a determinare il fatto che esso rappresenti un riferimento rvalue o lvalue. Se l’inizializzatore è un rvalue, il riferimento universale corrisponde a un riferimento rvalue. Se l’inizializzatore è un lvalue, il riferimento universale corrisponde a un riferimento lvalue. Per i riferimenti universali che sono parametri di funzione, l’inizializzatore viene fornito al momento della chiamata:

```
template<typename T>
void f(T&& param);           // param è un riferimento universale

Widget w;
f(w);                        // lvalue passato a f; il tipo di param
                             // è Widqet& (ovvero un riferimento
                             lvalue)

f(std::move(w) );            // rvalue passato a f; il tipo di param
                             // è Widqet&& (ovvero un riferimento
                             rvalue)
```

Perché un riferimento possa essere universale, la deduzione del tipo è necessaria, ma non sufficiente. Anche la *forma* della dichiarazione del riferimento deve essere corretta e tale forma deve sottostare a precisi vincoli. Deve essere esattamente `T&&`. Osservate nuovamente questo esempio basato sul codice che abbiamo visto in precedenza:

```
template<typename T>
void f(std::vector<T>&& param);   // param è un riferimento rvalue
```

Quando viene richiamata `f`, il tipo `T` viene dedotto (a meno che il chiamante non lo specifichi esplicitamente, un caso limite di cui non ci occuperemo). Ma la

forma della dichiarazione del tipo di `param` non è `T&&`, bensì `std::vector<T>&&`. Ciò elimina la possibilità che `param` sia un riferimento universale. Pertanto `param` sarà un riferimento rvalue, qualcosa che il compilatore sarà felice di comunicarvi se tenterete di passare a `f` un lvalue:

```
std::vector<int> v;
f(v);                    // errore! Non si può associare un lvalue
                         // a un riferimento rvalue
```

Anche la semplice presenza del qualificatore const è sufficiente per impedire che un riferimento possa essere universale:

```
template<typename T>
void f(const T&& param);   // param è un riferimento rvalue
```

Se vi trovate in un template e vedete un parametro di funzione di tipo `T&&`, potete essere portati a supporre che si tratti di un riferimento universale. Ma non è così. Il fatto che sia un template non garantisce la deduzione del tipo. Considerate la seguente funzione membro `push_back` in std::vector:

```
template<class T, class Allocator =
allocator<T>>                                   // dal C++
class vector {                                  // Standard
public:
   void push back(T&& x);
   …
};
```

Il parametro di `push_back` ha certamente la forma corretta per un riferimento universale, ma in questo caso non vi è la deduzione del tipo. Questo perché `push_back` non può esistere senza una particolare istanziazione di `vector` di cui possa far parte e il tipo di tale istanziazione determina completamente la dichiarazione per `push_back.` Pertanto, dicendo

```
std::vector<Widget> v;
```

si fa in modo che il template `std::vector` venga istanziato nel seguente modo:

```
class vector<Widget, allocator<Widget>>
{
public:
   void push back(Widget&& x);           // riferimento rvalue
   …
};
```

Ora si può vedere chiaramente che `pushback` non impiega la deduzione del tipo. Questa `push_back` per `vector<T>` (ve sono due, la funzione subisce overloading) dichiara sempre un parametro di tipo riferimento rvalue a `T`.

Al contrario, la funzione membro `emplace_back`, concettualmente simile, in `std::vector` *impiega* la deduzione del tipo:

```
template<class T, class Allocator =
allocator<T>>                            // sempre dal
class vector {                           // C++
public:                                  // Standard
   template <class... Args>
   void emplace back(ArgsSS... args);
   …
};
```

Qui, il parametro di tipo `Args` è indipendente dal parametro del tipo, `T`, di `vector` e pertanto `Args` deve essere dedotto ogni volta che viene richiamata `emplace_back` (d'accordo, in realtà `Args` non è esattamente un parametro per il tipo, ma per gli scopi di questa discussione, possiamo trattarlo come tale).

Il fatto che il parametro per il tipo di `emplace_back` si chiami `Args` e che comunque sia un riferimento universale va a supporto del mio precedente commento: che è la *forma* di un riferimento universale che deve essere `T&&`. Non è obbligatorio utilizzare il nome `T`. Per esempio, il seguente template prende un riferimento universale, poiché la forma `(type&&)` è corretta e il tipo di `param` verrà dedotto (nuovamente, escludendo il caso limite in cui il chiamante specifichi esplicitamente il tipo):

```
template<typename MyTemplateType>        // param è un
```

```
void someFunc(MyTemplateType&& param);   // riferimento universale
```

Ho detto in precedenza che anche le variabili `auto` possono essere riferimenti universali. Per essere più precisi, le variabili dichiarate di tipo `auto&&` sono riferimenti universali, poiché a esse si applica la deduzione del tipo e hanno la forma corretta (`T&&`). I riferimenti universali `auto` non sono altrettanto comuni dei riferimenti universali utilizzati per i parametri dei template di funzione, ma ogni tanto vengono impiegati in C++11. Sono invece più presenti in C++14, perché le espressioni lambda del C++14 possono dichiarare parametri `auto&&`. Per esempio, se voleste scrivere una lambda C++14 per registrare il tempo impiegato nella chiamata di una funzione arbitraria, potreste fare nel seguente modo:

```
auto timeFuncInvocation =
[](auto&& func, auto&&... params)          // C++14
{
   start timer;
   std::forward<decltype(func)>(func)(     // richiama func
    std::forward<decltype(params)>
    (params)...                            // su params
    );
   stop timer and record elapsed time;
};
```

Se la vostra reazione al codice `std::forward<decltype(`*bla bla bla)*> all'interno della lambda è "Ma che diamine...?!", questo significa che probabilmente non avete ancora letto l'Elemento 33. Non preoccupatevi. L'importante in questo Elemento è costituito dai parametri `auto&&` dichiarati dalla lambda. `func` è un riferimento universale che può essere associato a ogni oggetto richiamabile, che sia lvalue o rvalue. `args` è costituito da zero o più riferimenti universali (ovvero un *pack)* che può essere associato a un qualsiasi numero di oggetti di tipo arbitrario. Il risultato, grazie ai riferimenti universali `auto`, è che `timeFuncInvocation` può controllare l'esecuzione di *praticamente ogni* funzione (per dettagli su questa differenza fra "ogni" e "praticamente ogni", consultate l'Elemento 30).

Tenete in considerazione che questo intero Elemento (alla base dei riferimenti

universali) è una bugia... ops, volevo dire una “astrazione”. La verità su cui essa si basa è chiamata collasso dei riferimenti *(reference collapsing)*, un argomento al quale è dedicato l’Elemento 28. Ma la verità non rende meno utile l’astrazione. Il fatto di distinguere tra riferimenti rvalue e riferimenti universali vi aiuterà a leggere il codice sorgente con maggiore precisione (“Il `T&&` che ho davanti si associa solo a rvalue o a qualsiasi cosa?”) e vi eviterà di comunicare in modo ambiguo con i colleghi (“Sto utilizzando un riferimento universale, non un riferimento rvalue...”). Aiuterà inoltre a comprendere gli Elementi 25 e 26, che contano su questa distinzione. Accettate dunque questa astrazione. Come le leggi sulla gravitazione di Newton (tecnicamente errate) sono altrettanto utili e più facili da applicare rispetto alla teoria della relatività generale di Einstein (la “verità”), così il concetto di riferimento universale è normalmente preferibile a considerare i dettagli del “collasso”.

Argomenti da ricordare

- Se un parametro di un template di funzione ha tipo `T&&` per un tipo dedotto `T` o se un oggetto è dichiarato utilizzando `auto&&`, il parametro o l’oggetto è un riferimento universale.
- Se la forma della dichiarazione del tipo non è esattamente **type&&** o se la deduzione del tipo non si verifica, **type&&** identifica un riferimento rvalue.
- I riferimenti universali corrispondono a riferimenti rvalue se vengono inizializzati tramite un rvalue. Corrispondono invece a riferimenti lvalue se vengono inizializzati con un lvalue.

Elemento 25 – Usare std::move sui riferimenti rvalue e std::forward sui riferimenti universali

I riferimenti rvalue si associano solo a oggetti che sono candidati per lo spostamento. Se avete un parametro riferimento rvalue, sapete che l’oggetto cui si associa può essere spostato:

```
class Widget {
   Widget(WidgetSS rhs);            // rhs fa assolutamente riferimento
   …                                // a un oggetto spostabile
};
```

Per questo motivo, vorrete passare tali oggetti ad altre funzioni in un modo che permetta a tali funzioni di sfruttare il fatto che l'oggetto è un rvalue. Il modo per farlo consiste nel convertire in rvalue i parametri associati a tali oggetti. Come descrive l'Elemento 23, questo non è solo ciò che std::move fa, è anche il motivo stesso per cui è stata creata:

```
class Widget {
public:
   Widget(Widget&& rhs)                 // rhs è un riferimento rvalue
   : name(std::move(rhs.name)),
   p(std::move(rhs.p))
   { ... }
   …
private:
   std::string name;
   std::shared_ptr<SomeDataStructure> p;
};
```

Un riferimento universale, al contrario (vedi Elemento 24), *può essere associato* a un oggetto che può essere impiegato per uno spostamento. I riferimenti universali devono essere convertiti in rvalue solo se sono stati inizializzati tramite un rvalue. L'Elemento 23 spiega che questo è esattamente ciò che fa `std::forward`:

```
class Widget {
public:
   template<typename T>
```

```
    void setName(T&& newName)             // newName è un
    { name = std::forward<T>(newName); }  // riferimento universale

    …
};
```

In breve, i riferimenti rvalue dovrebbero essere *convertiti incondizionatamente* in rvalue (tramite `std::move`) quando vengono inoltrati ad altre funzioni, poiché sono *sempre* associati a rvalue; i riferimenti universali dovrebbero essere *convertiti condizionatamente* in rvalue (tramite `std::forward`) al momento dell'inoltro, poiché solo *talvolta* sono associati a rvalue.

L'Elemento 23 spiega che l'uso di `std::forward` su riferimenti rvalue può anche esibire un comportamento corretto, ma il codice sorgente diviene prolisso, soggetto a errori e di difficile comprensione; pertanto si dovrebbe evitare di utilizzare `std::forward` con riferimenti rvalue. Ancora peggiore è l'idea di utilizzare `std::move` con i riferimenti universali, poiché possono avere l'effetto di modificare inaspettatamente gli lvalue (ovvero le variabili locali):

```
class Widget {
public:
    template<typename T>
    void setName(T&& newName)             // riferimento universale
    { name = std::move(newName); }        // compilabile, ma
    …                                     // decisamente cattiva
                                          programmazione!

private:
    std::string name;
    std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName();              // funzione factory

Widget w;
```

```
auto n = getWidgetName();       // n è una variabile
                                // locale
w.setName(n);                   // sposta n in w!
…                               // il valore di n ora è
                                // ignoto
```

Qui, la variabile locale `n` viene passata a `w.setName`; il chiamante può erroneamente supporre che si tratti di un'operazione di sola lettura su `n`. Ma poiché `setName` usa internamente `std::move` per convertire incondizionatamente il suo parametro riferimento in un rvalue, il valore di `n` verrà spostato in `w.name` e dopo la chiamata a `setName`, `n` avrà un valore non specificato. Questo è un tipo di comportamento che genera collera nel chiamante.

Si potrebbe supporre che il parametro di `setName` non dovrebbe essere dichiarato come riferimento universale. Tali riferimenti non possono essere `const` (vedi Elemento 24), e certamente `setName` non dovrebbe certo modificare il proprio parametro. Potreste pensare che se `setName` avesse semplicemente subito overloading per lvalue `const` e per rvalue, tutto questo problema avrebbe potuto essere evitato. Nel seguente modo:

```
class Widget {
public:
   void setName(const std::string&        // impostato da
   newName)
   { name = newName; }                    // const lvalue

   void setName(std::string&& newName)    // impostato da
   { name = std::move(newName); }         // rvalue
   …
};
```

Questo certamente potrebbe funzionare (in questo caso), ma presenta dei problemi. Innanzitutto, vi è ulteriore codice sorgente da scrivere e correggere (due funzioni invece di un unico template). In secondo luogo, può essere meno efficiente. Per esempio, considerato il seguente uso di `setName`:

```
w.setName("Adela Novak");
```

Con la versione di `setName` che accetta un riferimento universale, a `setName` verrebbe passata la stringa letterale `"Adela Novak"`, la quale verrebbe inviata all'operatore di assegnamento per la `std::string` all'interno di `w`. Il dato membro `w` di `name` verrebbe pertanto assegnato direttamente dal letterale stringa; non ne deriverebbero oggetti `std::string` temporanei. Con le versioni in overloading di `setName`, invece, verrebbe creato un oggetto `setName` temporaneo cui associare il parametro di `setName`, e questa `std::string` temporanea verrebbe poi spostata nel dato membro di `w`. Una chiamata a `setName` richiederebbe, pertanto, l'esecuzione di: un costruttore di `std::string` (per creare la stringa temporanea), un operatore di assegnamento per spostamento di `std::string` (per spostare `newName` in `w.name`) e un distruttore di `std::string` (per distruggere la stringa temporanea). Questa è quasi certamente una sequenza di esecuzione più costosa rispetto alla semplice chiamata dell'operatore di assegnamento di `std::string`, che accetta un puntatore `const char*`. Il costo aggiuntivo varia da implementazione a implementazione e quanto occorra preoccuparsi di questo costo varia da applicazione ad applicazione e da libreria a libreria, ma il fatto è che la sostituzione di un template che accetta un riferimento universale con una coppia di funzioni in overloading per riferimenti lvalue e riferimenti rvalue, molto probabilmente prevedrà dei costi runtime. Se poi generalizziamo l'esempio, in modo che il dato membro di `Widget` possa essere di un tipo arbitrario (ovvero non abbiamo la garanzia che si tratti di una `std::string`), il divario prestazionale può ampliarsi notevolmente, poiché non tutti i tipi sono "economici" da spostare quanto le `std::string` (vedi Elemento 29).

Il problema più serio con l'overloading per lvalue e rvalue, tuttavia, non riguarda la quantità o la comprensibilità del codice sorgente, e neppure le prestazioni runtime del codice. È la cattiva scalabilità del progetto. `Widget::setName` accetta un solo parametro e pertanto richiede due soli overload, ma per le funzioni che richiedono più parametri, ognuno dei quali potrebbe essere un lvalue o un rvalue, il numero di overloading cresce geometricamente: con *n* parametri saranno necessari 2^n overload. Ma c'è di peggio. Alcune funzioni (in realtà template di funzioni) accettano un numero *illimitato* di parametri, ognuno dei quali potrebbe essere un lvalue o un rvalue. Il classico esempio è rappresentato da `std::make_shared` e, in C++14, da `std::make_unique` (vedi Elemento 21). Osservate le dichiarazioni degli overload utilizzati più

comunemente:

```
template<class T, class... Args>          // dal C++11
shared ptr<T> make shared(ArgsSS...       // Standard
args);

template<class T, class... Args>          // dal C++14
unique_ptr<T> make unique(Args&&...       // Standard
args);
```

Per funzioni come queste, l'overloading per lvalue e rvalue non è possibile: i riferimenti universali sono l'unico modo di procedere. E all'interno di tali funzioni, ve lo garantisco, ai parametri riferimento universale passati ad altre funzioni viene applicata `std::forward`. Questo è esattamente ciò che dovreste fare.

Almeno di solito. Talvolta. Ma non necessariamente per principio. In alcuni casi, intendete utilizzare l'oggetto associato a un riferimento rvalue o a un riferimento universale più di una volta in un'unica funzione e volete assicurarvi che non venga spostato finché non avrete terminato di utilizzarlo. In tal caso, dovrete applicare `std::move` (per i riferimenti rvalue) o `std::forward` (per i riferimenti universali) solo all'ultimo utilizzo del riferimento. Per esempio:

```
template<typename T>                 // text è
void setSignText(T&& text)           // rif. univ.
{
   sign.setText(text);               // usa text, ma
                                     // senza modificarlo
   auto now =                        // ottiene l'ora corrente
    std::chrono::system
    clock::now();

   signHistory.add(now,
      std::forward<T>(text));        // conversione condizionale
}                                    // di text in rvalue
```

Qui vogliamo assicurarci che il valore di `text` non venga cambiato da

`sign.setText`, poiché vogliamo utilizzare tale valore quando richiamiamo `signHistory.add`. Da qui l’uso di `std::forward` solo sull’utilizzo finale del riferimento universale.

Per `std::move` si applica lo stesso ragionamento (ovvero applicare `std::move` a un riferimento rvalue solo l’ultima volta che viene utilizzato), ma è importante notare che in alcuni rari casi si vuole richiamare `std::move_if_noexcept` invece di `std::move`. Per capire quando e perché, consultate l’Elemento 14.

Se siete in una funzione che restituisce *per valore* e state restituendo un oggetto associato a un riferimento rvalue o a un riferimento universale, dovrete applicare `std::move` o `std::forward` al momento del `return` del riferimento. Per capire perché, considerate una funzione `operator+` con la quale sommare due matrici, dove la matrice di sinistra si sa che è un rvalue (il suo spazio può pertanto essere riutilizzato per contenere la somma delle matrici):

```
Matrix                                     // return per-valore
operator+(MatrixSS lhs, const MatrixS
rhs)
{
   lhs += rhs;
   return std::move(lhs);                  // sposta lhs nel
}                                          // valore restituito
```

Convertendo lhs in un rvalue nell’istruzione return (tramite std::move), lhs verrà spostato nella posizione del valore restituito dalla funzione. Se la chiamata std::move venisse omessa,

```
Matrix                                     // come sopra
operator+(Matrix&& lhs, const Matrix&
rhs)
{
   lhs += rhs;
   return lhs;                             // copia lhs nel
}                                          // valore restituito
```

il fatto che `lhs` è un lvalue forzerebbe i compilatori a *copiarlo*, invece, nella posizione del valore restituito. Supponendo che il tipo `Matrix` supporti la

costruzione per spostamento, più efficiente rispetto alla costruzione per copia, l'uso di `std::move` nell'istruzione `return` genera codice più efficiente.

Se `Matrix` non supporta lo spostamento, la sua conversione in un rvalue non darà problemi, poiché tale rvalue verrà semplicemente copiato dal costruttore per copia di `Matrix` (vedi Elemento 23). Se però `Matrix` viene successivamente modificato per supportare lo spostamento, `operator+` se ne avvantaggerà automaticamente la prossima volta che verrà compilato. Per questo motivo, non si perderà nulla (anzi, si avrà un vantaggio) applicando `std::move` ai riferimenti rvalue restituiti dalle funzioni che restituiscono per valore.

La situazione è simile per i riferimenti universali e `std::forward`. Considerate un template di funzione `reduceAndCopy` che accetta un oggetto `Fraction` potenzialmente non ridotto, lo riduce e poi restituisce una copia del valore ridotto. Se l'oggetto originale è un rvalue, il suo valore dovrà essere spostato nel valore restituito (evitando così il costo dell'operazione di copia); ma se l'originale è un lvalue, deve essere eseguita una copia. Pertanto:

```
template<typename T>
Fraction                                 // restituisce per-valore
reduceAndCopy(T&& frac)                  // parametro riferimento
                                         universale
{
   frac.reduce();
   return std::forward<T>(frac);         // sposta rvalue nel
}                                        // valore restituito,
                                         copia lvalue
```

Se la chiamata a `std::forward` venisse omessa, `frac` verrebbe incondizionatamente copiato nel valore restituito da `reduceAndCopy`.

Alcuni programmatori usano questa informazione e tentano di estenderla anche alle situazioni in cui non si applica: "Se l'uso di `std::move` su un parametro riferimento rvalue da copiare nel valore restituito trasforma una costruzione per copia in una costruzione per spostamento", pensano, "posso eseguire la stessa operazione anche sulle variabili locali che restituisco". In altre parole, immaginano che, data una funzione che restituisce una variabile locale per

valore, come il seguente esempio,

```
Widget makeWidget()  // Cersione per “copia” di makeWidget
{
   Widget w;         // variabile locale

   …                 // configura w

   return w;         // “copia” w nel valore restituito
}
```

intenderebbero “ottimizzarla” trasformando la “copia” in uno spostamento:

```
Widget makeWidget()          // versione per spostamento di
                             makeWidget
{
   Widget w;
   …
   return std::move(w);      // sposta w nel valore restituito
}                            // (ASSOLUTAMENTE DA NON FARE!)
```

Tutte le virgolette e i commenti che ho inserito dovrebbero chiarire che questo ragionamento è errato. D’accordo, ma perché?

È errato poiché il Comitato di Standardizzazione è sempre un passo avanti ai programmatori quando si tratta di eseguire ottimizzazioni. È stato riconosciuto molto tempo fa che la versione per “copia” di `makeWidget` può evitare la necessità di copiare la variabile locale `w` costruendola nella memoria allocata per il valore che verrà restituito dalla funzione. Questa tecnica è chiamata *Return Value Optimization* (RVO) ed è espressamente suggerita dallo standard del C++ fin da quando esiste.

In realtà si vuole permettere tale *elisione della copia* solo nei punti in cui ciò non influenzi il comportamento osservabile del software. Per rendere più

comprensibile il linguaggio impiegato dallo Standard, i compilatori possono elidere la copia (o lo spostamento) di un oggetto locale[12] in una funzione che restituisce per valore se (1) il tipo dell'oggetto locale coincide con il tipo restituito dalla funzione e (2) l'oggetto locale e ciò che viene restituito.

Detto questo, osservate nuovamente la versione per "copia" di `makeWidget`:

```
Widget makeWidget()              // versione per "copia" di
                                 makeWidget
{
   Widget w;
   …
   return w;                     // "copia" w nel valore
                                 restituito
}
```

Qui entrambe le condizioni sono soddisfatte e potete fidarvi se vi dico che per questo codice, ogni compilatore C++ degno di questo nome impiegherà l'ottimizzazione RVO per evitare di copiare `w`. Questo significa che la versione per "copia" di `makeWidget`, in realtà, non copia nulla.

La versione per spostamento di `makeWidget` fa esattamente ciò che dice il nome (supponendo che Widget offra un costruttore per spostamento): sposta il contenuto di `w` nella posizione del valore restituito da `makeWidget`. Ma perché i compilatori non usano l'ottimizzazione RVO per eliminare lo spostamento, costruendo nuovamente `w` nella memoria allocata per il valore restituito dalla funzione? La risposta è semplice: non possono. La condizione (2) stabilisce che l'ottimizzazione RVO possa essere eseguita solo se ciò che viene restituito è un oggetto locale, ma questo non è ciò che la versione per spostamento di `makeWidget` sta facendo. Osservate nuovamente la sua istruzione `return`:

```
return std::move(w);
```

Ciò che viene restituito qui non è l'oggetto locale `w`, ma *un riferimento a w:* il risultato di `std::move(w)`. La restituzione di un riferimento a un oggetto locale non soddisfa le condizioni necessarie per l'ottimizzazione RVO, pertanto i compilatori devono spostare `w` nella posizione del valore restituito dalla funzione. Gli sviluppatori che cercano di aiutare i loro compilatori a eseguire

un’ottimizzazione applicando `std::move` a una variabile locale che viene restituita, stanno in realtà limitando le opzioni di ottimizzazione già disponibili per i loro compilatori!

Tuttavia la RVO è un’ottimizzazione. I compilatori non sono *obbligati* a elidere le operazioni di copia e spostamento, anche quando lo possono fare. I più paranoici possono temere che i compilatori vi puniranno per le operazioni di copia, per il semplice fatto che possono. O forse avete conoscenze sufficienti per capire che vi sono casi in cui l’ottimizzazione RVO è troppo complessa da implementare per i compilatori, per esempio quando diversi percorsi di controllo di una funzione restituiscono variabili locali differenti (i compilatori dovrebbero generare codice per costruire la variabile locale appropriata nella memoria destinata al valore restituito dalla funzione, ma come potrebbero i compilatori determinare quale variabile locale sarebbe appropriata?). In tal caso, potreste voler pagare il prezzo di uno spostamento, considerandolo un’assicurazione rispetto al costo di una copia. Pertanto potreste comunque considerare che è ragionevole applicare `std::move` a un oggetto locale che state restituendo, semplicemente perché sapete che non pagherete mai per una copia.

In tal caso, l’applicazione di `std::move` a un oggetto locale sarebbe *comunque* una cattiva idea. La parte dello Standard che suggerisce l’ottimizzazione RVO dice anche che se le condizioni di tale ottimizzazione sono verificate, ma i compilatori scelgono di non eseguire l’elisione dell’operazione di copia, l’oggetto restituito *deve essere trattato come un rvalue*. In pratica, lo Standard richiede che quando l’ottimizzazione RVO è permessa, o si svolge l’elisione della copia, oppure agli oggetti locali restituiti viene implicitamente applicata `std::move`. Pertanto, nella versione per “copia” di `makeWidget`,

```
Widget makeWidget()                        // come prima
{
   Widget w;
   …
   return w;
}
```

i compilatori devono elidere la copia di `w` o devono trattare la funzione come se fosse scritta nel seguente modo:

```
Widget makeWidget()
{
   Widget w;
   …
   return std::move(w);                    // tratta w come un rvalue, poiché
                                           // non è stata eseguita l'elisione della copia
}
```

La situazione è simile per i parametri di funzione passati per valore. In questo caso non è prevista l'elisione della copia per il valore restituito dalla funzione, ma, se vengono restituiti, i compilatori devono trattarli come rvalue. Come risultato, se il codice sorgente ha il seguente aspetto:

```
Widget makeWidget(Widget w)                // parametro per valore dello stesso
                                           // tipo del tipo restituito dalla funzione
{
   …
   return w;
}
```

i compilatori devono trattarlo come se fosse scritto nel seguente modo:

```
Widget makeWidget(Widget w)
{
   ...
return std::move(w);                       // tratta w come un rvalue
}
```

Ciò significa che se si usa `std::move` sull'oggetto locale restituito da una funzione che sta restituendo per valore, non potete aiutare i vostri compilatori (devono trattare l'oggetto locale come un rvalue se non eseguono l'elisione della

copia), ma certamente potete ostacolarli (impedendo l'ottimizzazione RVO). Vi sono situazioni in cui l'applicazione di `std::move` a una variabile locale può essere una scelta ragionevole (per esempio, quando la passate a una funzione e sapete che non utilizzerete più tale variabile), ma non per un'istruzione `return` che si qualificherebbe per un'ottimizzazione RVO o che restituisce un parametro per valore.

Argomenti da ricordare

- Applicate `std::move` ai riferimenti rvalue e `std::forward` ai riferimenti universali solo l'ultima volta che vengono utilizzati.
- Fate la stessa cosa per i riferimenti rvalue e per i riferimenti universali restituiti dalle funzioni che restituiscono per valore.
- Non applicate mai `std::move` o `std::forward` agli oggetti locali ottimizzabili con RVO.

Elemento 26 – Evitare l'overloading sui riferimenti universali

Supponete di dover scrivere una funzione che accetta come parametro un nome, registra la data e l'ora, poi aggiunge il nome a una struttura dati globale. Il risultato potrebbe essere una funzione con il seguente aspetto:

```
std::multiset<std::string> names; // struttura dati globale

void logAndAdd(const std::string&
name)
{
   auto now =                     // ottiene l'ora corrente
    std::chrono::system
    clock::now();

   log(now, "logAndAdd");         // crea la voce log
```

```
   names.emplace(name);            // aggiunge il nome alla struttura
}                                  // dati globale; vedi Elemento 42
                                   // per info su emplace
```

Non è codice da “buttare via”, ma non è efficiente quanto dovrebbe. Considerate tre potenziali chiamate:

```
std::string petName(“Darla”);

logAndAdd(petName);                        // passa std::string lvalue

logAndAdd(std::string(“Persephone”));      // passa std::string rvalue

logAndAdd(“Patty Dog”);                    // passa una stringa letterale
```

Nella prima chiamata, il parametro `name` di `logAndAdd` viene associato alla variabile `petName`. In `logAndAdd`, `name` viene poi passato a `names.emplace`. Poiché `name` è un lvalue, viene copiato in `names`. Non vi è alcun modo per evitare la copia, poiché a `logAndAdd` è stato passato un lvalue `(petName)`.

Nella seconda chiamata, il parametro `name` viene associato a un rvalue (la `std::string` temporanea creata esplicitamente da `“Persephone”)`. `name` stesso è un lvalue, pertanto viene copiato in `names`, ma riconosciamo che, in linea di principio, il suo valore potrebbe essere spostato in `names`. In questa chiamata, paghiamo per una copia, ma potremmo riuscire a ottenere il tutto utilizzando solo uno spostamento.

Nella terza chiamata, il parametro `name` viene nuovamente associato a un rvalue, ma questa volta si tratta di una `std::string` temporanea, che viene implicitamente creata da `“Patty Dog”`. Come nella seconda chiamata, tutti i `name` vengono copiati in `names`, ma in questo caso l’argomento originariamente passato a `emplace` è una stringa letterale. Se la stringa letterale fosse stata passata

direttamente a `emplace`, non vi sarebbe stata alcuna necessità di creare una `std::string` temporanea. Al contrario, `emplace` avrebbe usato la stringa letterale per creare l'oggetto `std::string` direttamente all'interno di `std::multiset`. In questa terza chiamata, quindi, paghiamo per copiare una `std::string`, ma in realtà non vi è alcun motivo di pagare neppure per uno spostamento, tantomeno per una copia.

Possiamo eliminare le inefficienze nella seconda e nella terza chiamata riscrivendo `logAndAdd` in modo che accetti un riferimento universale (vedi Elemento 24) e, sulla base dell'Elemento 25, applicare `std::forward` per inviare questo riferimento a `emplace`. Il risultato parla da solo:

```
template<typename T>
void logAndAdd(T&& name)
{
   auto now = std::chrono::system clock::now();
   log(now, “logAndAdd”);
   names.emplace( std::forward<T>( name)
   );
}
std::string petName(“Darla”);            // come prima

logAndAdd( petName);                     // come prima, copia
                                         // lvalue in multiset

logAndAdd( std::string(“Persephone”) ); // sposta rvalue invece
                                         // di copiarlo

logAndAdd(“Patty Dog”);                  // crea std::string
                                         // in multiset invece
                                         // di copiare una
std::string
                                         // temporanea
```

Efficienza perfetta!

Se il discorso si fermasse qui, potremmo fermarci e cambiare argomento, ma non abbiamo ancora parlato del fatto che i client non sempre hanno un accesso diretto ai nomi richiesti da `logAndAdd`. Alcuni client hanno solo un indice, utilizzato da `logAndAdd` per ricercare il nome corrispondente all'interno di una tabella. Per supportare tali client, `logAndAdd` subisce overloading:

```
std::string nameFromIdx(int idx);        // restituisce il nome
                                         // corrispondente a idx

void logAndAdd(int idx)                  // nuovo overload
{
   auto now =
   std::chrono::system_clock::now();
   log(now, “logAndAdd”);
   names.emplace(nameFromIdx(idx));
   }
```

La risoluzione delle chiamate ai due overloading funziona come previsto:

```
std::string petName(“Darla”);         // come prima

logAndAdd(petName);                   // come prima, tutte
logAndAdd(std::string(“Persephone”)); // queste chiamate richiamano
logAndAdd(“Patty Dog”);               // l’overload per T&&

logAndAdd(22);                        // richiama l’overload per
                                      int
```

In effetti, la risoluzione funziona come previsto solo se non ci si aspetta troppo. Supponete che un client abbia uno `short` che contiene un indice e che lo passi a `logAndAdd`:

```
short nameIdx;
…                                        // dare un valore a
                                         nameIdx
```

```
logAndAdd(nameIdx);                    // errore!
```

Il commento sull'ultima riga non è particolarmente illuminante, ecco dunque che cosa succede.

Vi sono due overload per `logAndAdd`. Quello che accetta un riferimento universale può dedurre `T` come `short`, trovando pertanto una corrispondenza esatta. L'overloading per un parametro `int` può corrispondere all'argomento `short` solo dopo una promozione. In base alle classiche regole di risoluzione dell'overloading, una corrispondenza esatta batte una corrispondenza per promozione, pertanto viene richiamato l'overloading per il riferimento universale.

All'interno di tale overloading, il parametro `name` viene associato allo `short` che gli viene passato. `name` viene poi inoltrato con `std::forward` alla funzione membro `emplace` su `names` (una `std::multiset<std::string>`), che, a sua volta, lo inoltra diligentemente al costruttore di `std::string`. Ma non esiste alcun costruttore per `std ::string` che accetti uno `short`, pertanto la chiamata al costruttore di `std::string` all'interno della chiamata `multiset::emplace` all'interno della chiamata a `logAndAdd` non ha successo. Tutto perché l'overloading per il riferimento universale era una corrispondenza migliore per un argomento `short` rispetto a un `int`.

Le funzioni che accettano riferimenti universali sono fra le più "fameliche" del C++. Si istanziano per creare corrispondenze esatte per quasi ogni tipo di argomento (i pochi argomenti che le "sfuggono" sono descritti nell'Elemento 30). Questo è il motivo per cui combinare l'overloading e i riferimenti universali è quasi sempre una cattiva idea: l'overloading per riferimenti universali si aggiudica molti più tipi di argomenti rispetto a quanti lo sviluppatore generalmente si immagini.

Un modo semplice per schivare questa trappola consiste nello scrivere un costruttore perfect-forward. Una piccola modifica all'esempio `logAndAdd` illustra il problema. Invece di scrivere una funzione che può prendere una `std::string` o un indice che possa essere utilizzato per ricercare una `std::string`, immaginate una classe `Person` con dei costruttori che facciano la stessa cosa:

```
class Person {
public:
```

```
    template<typename T>
    explicit Person(T&& n)                // costr. per perfect-
                                          // forward;
    : name(std::forward<T>(n)) {}         // inizializza il dato
                                          // membro

    explicit Person(int idx)              // costr. per int
    : name(nameFromIdx(idx)) {}
    …

private:
    std::string name;
};
```

Come nel caso di `logAndAdd`, il passaggio di un tipo intero diverso da `int` (ovvero `std::size_t`, `short`, `long` e così via) richiamerà l'overloading del costruttore per riferimenti universali invece che per `int` e ciò porterà a un errore di compilazione. Tuttavia, qui il problema è molto peggiore, poiché in `Person` vi sono molti più overloading rispetto a quanto si possa immaginare. L'Elemento 17 spiega che in condizioni appropriate, il C++ genererà i costruttori per copia e per spostamento e questo vale anche se la classe contiene un costruttore *template-izzato* che possa essere istanziato per produrre la signature del costruttore per copia o per spostamento. Se i costruttori per copia e per spostamento di `Person` vengono generati in questo modo, `Person` avrà il seguente aspetto:

```
class Person {
public:
    template<typename T>                  // costr. per perfect-
                                          // forward
    explicit Person(T&& n)
    : name(std::forward<T>(n)) {}

    explicit Person(int idx);             // costr. per int

    Person(const Persons rhs);            // costr. per copia
                                          // (generato dal
```

```
                                        compilatore)
   Person(Person&& rhs);                // costr. per spostamento
                                        // (generato dal
   …                                    compilatore)
};
```

Questo porta a un comportamento che può essere intuitivo solo per chi ha lavorato sui compilatori, tanto da aver dimenticato la sua essenza umana.

```
Person p("Nancy");

auto cloneOfP(p);    // crea una nuova Person da p;
                     // incompilabile!
```

Qui stiamo tentando di creare una `Person` da un'altra `Person`, che sembra un caso davvero ovvio di costruzione per copie (`p` è un lvalue e così possiamo fugare tutti i dubbi che possiamo avere sul fatto che venga eseguita una "copia" attraverso un'operazione di spostamento). Ma questo codice non richiamerà il costruttore per copie. Richiamerà il costruttore perfect-forward. Tale funzione tenterà poi di inizializzare il dato membro `std::string` di `Person` con un oggetto di tipo `Person (p)`. Poiché `std::string` non ha alcun costruttore che accetta un `Person`, il compilatore alzerà le mani esasperato, punendovi come sa fare: con lunghi e incomprensibili messaggi di errore per esprimere tutto il proprio disappunto.

"Perché", potreste chiedervi, "viene richiamato il costruttore per il perfect-forward invece del costruttore per copia? Stiamo inizializzando una `Person` con un'altra `Person!"`. In effetti è così, ma i compilatori sono soliti seguire le regole del C++ e le regole impiegate qui si occupano proprio di governare la risoluzione delle chiamate alle funzioni in overloading.

Il compilatore ragiona nel seguente modo. `cloneOfP` viene inizializzato con un lvalue non-**`const (p)`** e ciò significa che il costruttore *template-izzato* può essere distanziato per prendere un lvalue non-**`const`** di tipo `Person`. Dopo tale istanziazione, la classe `Person` ha il seguente aspetto:

```
class Person {
```

```
public:
   explicit Person(Person& n)            // istanziata dal
   : name(std::forward<Person&>(n)) {}   // template
                                         // per perfect-forward

   explicit Person(int idx);             // come prima
   Person(const Persons rhs);            // costr. per copia
                                         // (generato dal
                                         compilatore)
   };
```

Nell'istruzione,

```
auto cloneOfP(p);
```

p potrebbe essere passato al costruttore per copia o al template istanziato. La chiamata del costruttore per copia richiederebbe l'aggiunta di `const` a `p` per coincidere con il tipo di parametri del costruttore per copia, mentre la chiamata del template istanziato non richiede questa aggiunta. L'overloading generato dal template è pertanto una corrispondenza migliore e così i compilatori faranno ciò che sono progettati per fare: generare una chiamata alla funzione che offre la migliore corrispondenza. La "copia" di lvalue non**-const** di tipo `Person` viene pertanto gestita dal costruttore per perfect-forward, non dal costruttore per copia.

Se cambiamo leggermente l'esempio, in modo che l'oggetto da copiare sia `const`, la situazione è completamente differente:

```
const Person cp("Nancy");            // ora l'oggetto è const

auto cloneOfP(cp);                   // richiama il costr. per
                                     copia!
```

Poiché ora l'oggetto da copiare è `const`, si tratta di una corrispondenza esatta per il parametro del costruttore per copia. Il costruttore template-izzato può essere istanziato in modo da avere la stessa signature,

```
class Person {
```

```
public:
  explicit Person(const Persons n);     // istanziato dal
                                        // template

  Person(const Person& rhs);            // costr. per copia
                                        // (generato dal
                                        compilatore)
  …
};
```

ma questo non conta, poiché una delle regole di risoluzione degli overloading in C++ è che nelle situazioni in cui un'istanziazione a template e una funzione non-template (ovvero una normale funzione) rappresentano corrispondenze ugualmente buone per una chiamata a funzione, dev'essere preferita la funzione "normale". Il costruttore per copia (una funzione normale) batte pertanto un template istanziato con la stessa signature.

Se vi state chiedendo perché i compilatori generino un costruttore per copia quando potrebbero istanziare un costruttore template-izzato per ottenere la signature che avrebbe il costruttore per copia, rileggete l'Elemento 17.

L'interazione fra i costruttori perfect-forward e le operazioni di copia e spostamento generate dal compilatore si complica ulteriormente quando entra in gioco l'ereditarietà. In particolare, le implementazioni convenzionali delle operazioni di copia e spostamento per le classi derivate hanno comportamenti piuttosto sorprendenti.

Ecco un esempio:

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson&    // costr. per copia;
  rhs)                                  richiama
  : Person(rhs)                         // il costr. perfect-
                                        forward
  { ... }                               // della classe base!

  SpecialPerson(SpecialPerson&& rhs)    // costr. per spostamento;
                                        richiama
```

```
    : Person(std::move(rhs))          // il costr. perfect-forward
    { ... }                           // della classe base!
};
```

Come indicato nel commento, i costruttori per copia e per spostamento della classe derivata non richiamano i costruttori per copia e per spostamento della classe base, bensì il costruttore perfect-forward della classe base! Per comprendere il perché, notate che le funzioni delle classi derivate stanno usando argomenti di tipo `SpecialPerson`, da passare alla loro classe base, quindi sono sensibili alle conseguenze dell'istanziazione template e della risoluzione dell'overloading dei costruttori della classe `Person`. Alla fine, il codice non verrà compilato, poiché non esiste un costruttore per `std::string` che accetta una `SpecialPerson`.

Spero, con questo, di avervi convinto che l'overloading per parametri che sono riferimenti universali dovrebbe essere evitato il più possibile. Ma se l'overloading per riferimenti universali è una cattiva idea, cosa fare quando occorre una funzione che inoltri la maggior parte dei tipi di argomenti, ma bisogna comunque trattare alcuni tipi di argomenti in modo particolare? Il problema può essere risolto in vari modi. Talmente tanti che gli abbiamo dedicato l'intero Elemento 27. Dunque vi basta continuare a leggere.

Argomenti da ricordare

- L'overloading per riferimenti universali quasi sempre fa sì che questo venga richiamato più frequentemente del dovuto.
- I costruttori perfect-forward sono particolarmente problematici, poiché, in genere, sono corrispondenze migliori rispetto ai costruttori per copia per gli lvalue non-**`const`** e possono attirare le chiamate dalla classe derivata ai costruttori per copia e spostamento della classe base.

Elemento 27 – Alternative all'overloading per riferimenti universali

L'Elemento 26 spiega che l'overloading per riferimenti universali può causare vari problemi, sia per le funzioni indipendenti sia per le funzioni membro (in particolare i costruttori). Inoltre, vi trovate esempi in cui tale overloading potrebbe essere utile. Se solo si comportasse nel modo che vorremmo! Questo Elemento esplora i modi per ottenere il comportamento desiderato, o tramite meccanismi che evitano di ricorrere all'overloading per riferimenti universali o impiegandoli in modi che vincolano i tipi di argomenti cui essi rispondono.

La discussione che segue sviluppa gli esempi introdotti nell'Elemento 26. Se non lo avete letto recentemente, può essere il caso di ripassarlo, prima di procedere.

Abbandonare l'overloading

Il primo esempio dell'Elemento 26, `logAndAdd`, è rappresentativo delle molte funzioni che aiutano ad aggirare i problemi dell'overloading per riferimenti universali utilizzando nomi differenti per quelli che dovrebbero essere overloading. I due overloading di `logAndAdd`, per esempio, possono essere suddivisi in `logAndAddName` e `logAndAddNameIdx`. Purtroppo, questo approccio non funziona per il secondo esempio che abbiamo considerato, il costruttore di `Person`, poiché il nome del costruttore viene corretto dal linguaggio. Ma, a parte questo, perché rinunciare completamente all'overloading?

Passaggio per const T&

Un'alternativa consiste nel tornare al C++98 e sostituire i passaggi per riferimento universale con altrettanti passaggi per riferimento lvalue a `const`. In realtà, questo è il primo approccio considerato nell'Elemento 26 (vedi a pagina 162). Il difetto di questo approccio è che il risultato non è efficiente come vorremmo. Sapendo ciò che sappiamo sull'interazione fra i riferimenti universali e l'overloading, rinunciare a un po' di efficienza per guadagnare in termini di semplicità può essere un'alternativa meno disdicevole di quanto possa sembrare.

Passaggio per valore

Un approccio che spesso consente di migliorare le prestazioni senza aumentare la complessità consiste nel sostituire, in modo assolutamente non intuitivo, i parametri a passaggio per riferimento con altrettanti passaggi per valore. Questa tecnica recepisce il consiglio fornito nell'Elemento 41 di considerare il passaggio di oggetti per valore quando si sa che essi verranno copiati; dunque si rimanda a tale Elemento per una discussione dettagliata del funzionamento di questo meccanismo e della sua efficienza. Qui vedremo solo come utilizzare la tecnica nell'esempio di `Person`:

```
class Person {
public:
   explicit Person(std::string n)      // sostituisce il costr. di T&& ctor;
   : name(std::move(n)) {}             // vedi Elemento 41 per l'uso di std::move

   explicit Person(int idx)            // come prima
   : name(nameFromIdx(idx)) {}
   …
private:
   std::string name;
};
```

Poiché non vi è alcun costruttore di `std::string` che accetta solo un intero, tutti gli argomenti `int` (o comunque di tipo simile a `int`) di un costruttore di `Person` (ovvero `std::size_t`, `short`, `long`) vengono incanalati nell'overloading per `int`. Analogamente, tutti gli argomenti di tipo `std::string` (e tutto ciò che da `std::string` può essere creato, come le stringhe letterali come `“Ruth”`) vengono passati al costruttore che accetta una `std::string`. Dunque nessuna sorpresa per i chiamanti. Potete però pensare, immagino, che qualcuno possa sorprendersi del fatto che utilizzando `0` o `NULL` per indicare un puntatore nullo si richiamerebbe l'overloading per `int`; ma, a questo punto, dovreste consultare l'Elemento 8 e continuare a leggerlo finché non fugherete ogni dubbio sull'uso di `0` o **NULL** come puntatori nulli.

L’approccio tag dispatch

Né il passaggio per riferimento lvalue a `const` né il passaggio per valore offrono il supporto per il perfect-forward. Se la motivazione per ricorrere a un riferimento universale è il perfect-forward, dobbiamo utilizzare un riferimento universale, non abbiamo altra scelta. Tuttavia non vogliamo abbandonare del tutto l’overloading. Pertanto, se non vogliamo rinunciare all’overloading e neppure ai riferimenti universali, come possiamo evitare l’overloading sui riferimenti universali?

In realtà non è così difficile. Le chiamate alle funzioni in overloading vengono risolte osservando tutti i parametri di tutti gli overloading, così come tutti gli argomenti nel punto della chiamata, scegliendo poi la funzione che rappresenta la corrispondenza migliore (tenendo in considerazione tutte le combinazioni parametro/argomento). Un parametro riferimento universale, generalmente, fornisce una corrispondenza esatta per tutto ciò che gli viene passato, ma se il riferimento universale fa parte di un elenco di parametri che contiene altri parametri che *non* sono riferimenti universali, alcune corrispondenze sufficientemente imprecise sui parametri che *non* sono riferimenti universali possono evitare che venga impiegato un overloading per riferimenti universali. Questo è ciò su cui si basa l’approccio *tag dispatch*. Un esempio aiuterà a comprendere la descrizione che segue.

Applicheremo la tecnica tag dispatch all’esempio `logAndAdd` presentato a pagina 162. Ecco il codice di tale esempio:

```
std::multiset<std::string> names;       // struttura dati globale

template<typename T>                    // crea la voce log e
                                        aggiunge
void logAndAdd(T&& name)                // name alla struttura
                                        dati
{
   auto now = std::chrono::system clock::now();
   log(now, “logAndAdd”);
   names.emplace(std::forward<T>(name));
}
```

Di per se stessa, questa funzione si comporta bene, ma dovendo introdurre l'overload che accetta un `int` utilizzato per ricercare gli oggetti per indice, ci ritroviamo alla situazione problematica dell'Elemento 26. L'obiettivo di questo Elemento consiste proprio nell'evitarla. Invece di aggiungere l'overload, reimplementeremo `logAndAdd` per delegare il lavoro ad altre due funzioni: una per i valori interi e una per tutto il resto. `logAndAdd` stessa accetterà tutti i tipi di argomenti, sia interi sia di altro tipo.

Le due funzioni che svolgono il lavoro vero e proprio si chiameranno `logAndAddImpl`, ovvero utilizzeremo l'overloading. Una delle funzioni accetterà un riferimento universale. Pertanto, avremo sia l'overloading sia i riferimenti universali. Ma entrambe le funzioni accetteranno anche un secondo parametro, che indica se l'argomento passato è di tipo intero. Questo secondo parametro è ciò che ci impedirà di precipitare nel problema della "voracità" descritto nell'Elemento 26, poiché faremo in modo che il secondo parametro determini quale overload selezionare.

Mi sembra di sentirvi brontolare: "Bla, bla, bla... Smettila di blaterare e mostraci il codice!". Nessun problema. Ecco una versione quasi-corretta di logAndAdd:

```
template<typename T>
void logAndAdd(T&& name)
{
   logAndAddImpl(std::forward<T>(name),
      std::is integral<T>());           // non correttissima
}
```

Questa funzione inoltra il proprio parametro a `logAndAddImpl`, ma passa anche un argomento che indica se il tipo di tale parametro (`T`) è intero. Quanto meno, questo è ciò che dovrebbe accadere. Per gli argomenti interi che sono rvalue, è anche ciò che succede. Ma, come descritto nell'Elemento 28, se al riferimento universale `name` viene passato un argomento lvalue, il tipo dedotto per `T` sarà un riferimento lvalue. Pertanto, se a `logAndAdd` viene passato un lvalue di tipo `int`, `T` verrà dedotto come `int&`. Questo non è un tipo intero, poiché i riferimenti non sono interi. Ciò significa che `std::is_integral<T>` sarà `false` per ogni argomento lvalue, anche se, in effetti, l'argomento rappresenta un valore intero.

Il fatto di aver individuato il problema ci consente di trovare una soluzione, poiché la sempre disponibile Libreria Standard del C++ ha un type trait (vedi Elemento 9), `std ::remove_reference`, che fa ciò che suggerisce il nome, ossia quello di cui abbiamo bisogno: rimuovere da un tipo ogni qualificatore di riferimento. Il modo corretto per scrivere `logAndAdd` è pertanto:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(
  std::forward<T>(name),
  std::is_integral<typename std::remove_reference<T>::type>()
  );
}
```

Questa forma è finalmente corretta (il C++14, è più compatto: si puà usare `std::remove_reference_t<T>` al posto del testo evidenziato; per i dettagli, consultate l'Elemento 9).

Tenendo conto di tutto questo, possiamo rivolgere l'attenzione alla funzione chiamata, `logAndAddImpl`. Vi sono due overload e il primo è applicabile solo ai tipi non interi (ovvero ai tipi per i quali `std::is_integral<typename std::remove_reference<T>::type>` dà `false`):

```
template<typename T>                             // non-integral
void logAndAddImpl(T&& name, std::false_type)    // argomento:
{                                                // lo aggiunge
   auto now =                                    // alla struttura
   std::chrono::system_clock::now();             dati
   log(now, "logAndAdd");                        // globale
   names.emplace(std::forward<T>(name));
}
```

Si tratta di codice piuttosto semplice, una volta compresi i meccanismi su cui si basa il parametro evidenziato. Teoricamente, `logAndAdd` passa un valore booleano a `logAndAddImpl`, indicando se a `logAndAdd`, è stato passato il tipo intero, ma `true` e `false` sono valori *runtime* e noi dobbiamo utilizzare una risoluzione degli overload (*un'attività di compilazione*) per scegliere l'overload corretto di `logAndAddImpl`. Ciò significa che abbiamo bisogno di un tipo che corrisponda a `true` e di un tipo differente che corrisponda a `false`. Questa

esigenza è talmente comune che la Libreria Standard fornisce ciò di cui c'è bisogno attraverso i nomi `std::true_type` e `std::false_type`. L'argomento passato a `logAndAddImpl` da `logAndAdd` è un oggetto di un tipo che eredita da `std::true_type` se `T` è intero e da `std::false_type` se `T` non è intero. Il risultato pratico è che questo overload di `logAndAddImpl` è un candidato utilizzabile per la chiamata in `logAndAdd` solo se `T` non è un tipo intero.

Il secondo overload copre il caso opposto: quando `T` è un tipo intero. In tal caso, `logAndAddImpl` trova semplicemente il nome corrispondente all'indice passato e poi passa il nome a logAndAdd:

```
std::string nameFromIdx(int idx);          // vedi Elemento 26
void logAndAddImpl(int idx,
std::true_type)                            // argomento
{                                          // intero: cerca
   logAndAdd(nameFromIdx(idx));            // il nome e
}                                          // richiamalogAndAdd
                                           // con tale nome
```

Facendo in modo che `logAndAddImpl` per un indice ricerchi il nome corrispondente e lo passi a `logAndAdd` (da cui verrà inoltrato con `std::forward` all'altro overload di `logAndAddImpl`), evitiamo la necessità di inserire il codice di log in entrambi gli overload di **`logAndAddImpl`.**

Utilizzando questa tecnica, i tipi `std::true_type` e `std::false_type` sono semplici "tag", il cui unico scopo è quello di forzare la risoluzione dell'overloading in modo che proceda esattamente come desideriamo. Notate che non assegniamo neppure un nome a questi parametri. Non hanno alcuno scopo runtime e, in realtà, speriamo che i compilatori riconosceranno che i parametri tag non vengono utilizzati e li ottimizzeranno escludendoli dall'immagine d'esecuzione del programma (alcuni compilatori lo fanno, ma non sempre). La chiamata alle funzioni di implementazione dell'overloading all'interno di `logAndAdd`, distribuisce (dispatch) il lavoro all'overload corretto, facendo sì che venga creato l'oggetto tag corretto. Da qui il nome di questa tecnica: *tag dispatch*. Si tratta di un elemento costitutivo standard della metaprogrammazione a template e più osserverete il codice delle librerie C++ di oggi, più lo incontrerete.

Per i nostri scopi, ciò che è importante relativamente alla tecnica tag dispatch

non è tanto come funziona, quanto come essa permetta di combinare riferimenti universali e overloading, eliminando il problema descritto nell'Elemento 26. La funzione di dispatch, `logAndAdd`, accetta un parametro riferimento universale non vincolato, ma questa funzione non subisce overloading. Le funzioni di implementazione, `logAndAddImpl`, subiscono invece overloading e una di esse accetta un parametro che è un riferimento universale, ma la risoluzione delle chiamate di queste funzioni dipende non solo dal parametro riferimento universale, ma anche dal parametro tag. E i valori del tag sono progettati in modo che un solo overload sia una corrispondenza utilizzabile. Il risultato è che è il tag a determinare quale overload richiamare. A questo punto, il fatto che il parametro riferimento universale generi sempre una corrispondenza esatta per il proprio argomento diviene ininfluente.

Vincolare i template che accettano riferimenti universali

Un elemento tipico della tecnica tag dispatch è l'esistenza di un'unica funzione (senza overloading) come API client. Quest'unica funzione distribuisce il lavoro alle funzioni di implementazione. La creazione di una funzione dispatch senza overload è normalmente semplice, ma il secondo problema considerato nell'Elemento 26, quello del costruttore perfect-forward della classe `Person` (vedi a pagina 165), è un'eccezione. I compilatori possono generare specifici costruttori per copia e per spostamento e pertanto anche scrivendo un solo costruttore e utilizzandovi la tecnica tag dispatch, alcune chiamate al costruttore potrebbero essere gestite da funzioni generate dal compilatore, che aggirano il nostro sistema tag dispatch.

In verità, il vero problema non è che le funzioni generate dal compilatore possano aggirare la struttura del tag dispatch, è che *questo non capita sempre.* Si desidera (praticamente sempre) che il costruttore per copia di una classe gestisca le richieste di copiare lvalue di tale tipo; ma, come illustra l'Elemento 26, fornendo un costruttore che accetta un riferimento universale accade che nella copia di lvalue non-**`const`** venga richiamato il costruttore per riferimenti universali e non il costruttore per copia. Tale Elemento spiega anche che quando una classe base dichiara un costruttore perfect-forward, tale costruttore verrà tipicamente richiamato quando le classi derivate implementano i propri costruttori per copia e per spostamento in modo convenzionale, anche se il comportamento corretto stabilirebbe che venissero richiamati i costruttori per copia e per spostamento della classe base.

Per situazioni di questo tipo, in cui una funzione in overloading che accetta un riferimento universale è più "vorace" di quanto dovrebbe, ma non abbastanza "aggressiva" da agire come unica funzione di dispatch, la tecnica tag dispatch non è più la soluzione più appropriata. Occorre impiegare una tecnica differente, che consenta di determinare le condizioni in cui possa essere impiegato il template di funzione di cui fa parte il riferimento universale. C'è bisogno di `std::enable_if`.

`std::enable_if` fornisce un modo per costringere i compilatori a comportarsi

come se un determinato template non esistesse. Tali template si dicono *disattivati*. Per default, tutti i template sono *attivi*, ma un template che utilizza `std::enable_if` è attivo solo se è soddisfatta la condizione specificata da `std::enable_if`. Nel nostro caso, vorremmo attivare il costruttore perfect-forward di `Person` solo se il tipo passato non è `Person`. Se, invece, il tipo passato è `Person`, vogliamo disattivare il costruttore perfect-forward (ovvero far sì che i compilatori lo ignorino), per far sì che la chiamata venga gestita dal costruttore per copia o per spostamento, che è ciò che vogliamo quando un oggetto `Person` viene inizializzato con un altro `Person`.

Il modo per esprimere questo concetto non è particolarmente difficoltoso, ma la sintassi non è proprio esaltante, specialmente le prime volte; pertanto cercheremo di semplificarla. Partiremo dalla condizione di `std::enable_if`. Ecco la dichiarazione per il costruttore perfect-forward di `Person`, che mostra solo quanto è necessario per poter utilizzare `std::enable_if`. Mostrerò solo la dichiarazione di questo costruttore, poiché l'uso di `std::enable_if` non ha alcun effetto sull'implementazione della funzione. L'implementazione rimane la stessa descritta nell'Elemento 26.

```
class Person {
public:
  template<typename T,
           typename = typename std::enable_if<condition>::type>
  explicit Person(T&& n);
  …
};
```

Per comprendere esattamente ciò che accade nel testo evidenziato, purtroppo devo rimandarvi ad altre fonti, poiché la descrizione dei dettagli richiederebbe troppo tempo. Nelle ricerche, utilizzate i termini "SFINAE" e "std::enable_if", poiché SFINAE è la tecnologia che consente il funzionamento di `std::enable_if`. Qui voglio concentrarmi sul modo in cui esprimere la condizione che controllerà se questo costruttore è attivo.

La condizione che vogliamo specificare è che `T` non sia `Person`, ovvero che il costruttore template-izzato debba essere attivato solo se `T` è un tipo diverso da `Person`. Grazie a un type trait che determina se due tipi coincidono (`std::is_same`), sembrerebbe che la condizione che vogliamo sia `!std::is_same<Person, T>::value` (notate il "`!`" all'inizio dell'espressione, perché vogliamo che `Person` e `T` *non* siano la stessa cosa). Questo è più o meno ciò che vogliamo, ma non è del tutto corretto, poiché, come descritto

nell'Elemento 28, il tipo dedotto per un riferimento universale inizializzato con un lvalue è sempre un riferimento lvalue. Ciò significa che per codice come il seguente,

```
Person p("Nancy");

auto cloneOfP(p);                        // inizializza da lvalue
```

il tipo `T` nel costruttore universale verrà dedotto come `Person&`. I tipi `Person` e `Person&` non sono la stessa cosa e il risultato di `std::is_same` rifletterà questo fatto: `std ::is_same<Person, Person&>::value` sarà quindi `false`.

Se riflettiamo bene su ciò che intendiamo quando diciamo che il costruttore template-izzato in `Person` debba essere attivato solo se `T` non è `Person`, ci accorgiamo che quando consideriamo `T` vogliamo ignorare i seguenti fatti.

- **Che sia un riferimento.** Per lo scopo di determinare se il costruttore per riferimenti universali debba essere attivato, i tipi `Person`, `Person&` e `Person&&` sono la stessa cosa di `Person`.

- **Che sia const o volatile.** Per quel che ci riguarda, una `const Person`, una `volatile Person` e una `const volatile Person` sono sempre una **`Person`.**

Ciò significa che abbiamo bisogno di un modo per eliminare da `T` tutti i riferimenti, i `const` e i `volatile` prima di controllare se il suo tipo è lo stesso di `Person`. Ancora una volta, la Libreria Standard ci dà ciò di cui abbiamo bisogno sotto forma di un type trait. Tale trait è `std::decay`. In pratica, `std::decay<T>::type` è la stessa cosa di `T`, tranne il fatto che sono stati rimossi i riferimenti e i qualificatori `const` e `volatile` *(qualificatori cv)*. In realtà le cose sono più complesse, poiché `std::decay`, come suggerisce il nome, trasforma anche gli array e i tipi funzione in puntatori (vedi Elemento 1), ma per gli scopi di questa discussione, `std::decay` si comporta come abbiamo descritto. La condizione che vogliamo impiegare per controllare se il nostro costruttore è attivato, pertanto, è

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

ovvero che `Person` non sia lo stesso tipo di `T`, ignorando i riferimenti e i qualificatori `const` e `volatile` (come descritto nell'Elemento 9, il `typename`

davanti a `std::decay` è obbligatorio, poiché il tipo `std::decay<T>::type` dipende dal parametro del template, `T)`.

Inserendo questa condizione in `std::enable_if` e formattando il risultato per renderlo più comprensibile, si ottiene questa dichiarazione del costruttore perfect-forward di `Person`:

```
class Person {
public:
  template<
  typename T,
  typename = typename std::enable_if<
         !std::is_same<Person,
                  typename std::decay<T>::type
              >::value
         >::type
  >
  explicit Person(T&& n);
  …
};
```

Se non avete mai visto niente di simile prima d'ora, siete davvero fortunati. C'è un motivo per cui ho presentato questa tecnica per ultima. Quando potete utilizzare uno degli altri meccanismi per evitare di impiegare insieme riferimenti universali e overloading (e quasi sempre è possibile), dovreste farlo. In ogni caso, una volta fatta l'abitudine alla sintassi funzionale e alla proliferazione di parentesi angolari, le cose non sono poi così complesse. Inoltre, questo è proprio il comportamento desiderato. Data la dichiarazione precedente, la costruzione di una `Person` da un'altra `Person` (che sia lvalue o rvalue, `const` o non**`-const`**, **`volatile`** o non**`-volatile)`** non richiamerà mai il costruttore che accetta un riferimento universale.

È davvero così? Davvero siamo a cavallo?

Veramente no. C'è una situazione evidenziata nell'Elemento 26 che continua a sfuggirci e dobbiamo occuparcene.

Supponete che una classe derivata da `Person` implementi le operazioni di copia e spostamento in modo convenzionale:

```
class SpecialPerson: public Person {
public:
   SpecialPerson(const SpecialPerson& rhs)   // costr. per copia; richiama
   : Person(rhs)                             // il costr. perfect-forward
   { ... }                                   // della classe base!

   SpecialPerson(SpecialPerson&& rhs)        // costr. per spostamento; richiama
   : Person(std::move(rhs))                  // il costr. perfect-forward
   { ... }                                   // della classe base!

};
```

Questo è lo stesso codice mostrato nell'Elemento 26 (a pagina 168), compresi i commenti, che, purtroppo, rimangono validi. Quando copiamo o spostiamo un oggetto `SpecialPerson`, ci aspettiamo di copiare o spostare le parti della sua classe base utilizzando i costruttori per copia e per spostamento della classe base. Ma, in queste funzioni, passiamo gli oggetti `SpecialPerson` ai costruttori della classe base, e poiché `SpecialPerson` non è la stessa cosa di `Person` (neppure dopo l'applicazione di `std:: decay`), il costruttore per riferimenti universali nella classe base è attivo e, se tutto va bene, si istanzia per trovare una corrispondenza esatta per un argomento `SpecialPerson`.

Questa corrispondenza esatta è migliore rispetto alle conversioni da derivata a base che sarebbero necessarie per associare gli oggetti `SpecialPerson` ai parametri `Person` nei costruttori per copia e per spostamento di `Person`. Pertanto, con il codice che abbiamo a questo punto, la copia e lo spostamento di oggetti `SpecialPerson` utilizzerebbero il costruttore perfect-forward di `Person` per copiare o spostare parti della classe base! Ci sembra di tornare all'Elemento 26.

La classe derivata sta semplicemente seguendo le normali regole per l'implementazione dei costruttori per copia e spostamento della classe derivata; pertanto la correzione di questo problema riguarda la classe base e, in

particolare, la condizione che controlla se è attivo il costruttore per riferimenti universali di `Person`. A questo punto comprendiamo che non vogliamo attivare il costruttore a template solo per un qualsiasi tipo di argomento diverso da `Person`, ma anche per ogni tipo di argomento diverso da Person *o anche da un tipo derivato* da `Person`. Dannata ereditarietà!

Non vi sorprenderete di sapere che fra i type trait standard ve ne è uno che determina se un tipo è derivato da un altro. Si chiama `std::is_base_of`. Dunque: `std::is_base_of<T1, T2>::value` è `true` se `T2` è un tipo derivato da `T1`. I tipi sono considerati derivati da se stessi e, pertanto, `std::is_base_of<T, T>::value` è `true`. Questo è comodo, poiché vogliamo modificare la condizione che controlla l'attivazione del costruttore perfect-forward di `Person` in modo che tale costruttore sia attivo solo se il tipo `T`, dopo aver eliminato tutti i riferimenti dei qualificatori `const` e `volatile`, non è né `Person` né una classe derivata da `Person`. Utilizzando `std::is_base_of` invece di `std::is_same`, otteniamo ciò di cui abbiamo bisogno:

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                !std::is_base_of<Person,
                       typename std::decay<T>::type
                        >::value
                 >::type
  >
  explicit Person(T&& n);
  …
};
```

Finalmente abbiamo concluso. Ammettendo di scrivere codice in C++11, le cose stanno così. Se invece scriviamo in C++14, questo codice funzionerà, ma possiamo impiegare template alias per `std::enable_if` e `std::decay`, per sbarazzarci del `typename` e del `::type`, ottenendo codice più lineare:

```
class Person {                                  // C++14
public:
  template<
   typename T,
    typename = std::enable if t<          // meno codice qui
```

```
                  !std::is_base_of<Person,
                         std::decay t<T>   // e qui
                        >::value
      >                                    // e qui
   >
   explicit Person(T&& n);
   …
};
```

Devo ammetterlo: vi ho mentito. Il discorso non è concluso, ma quasi ci siamo.

Abbiamo visto come utilizzare `std::enable_if` per disabilitare selettivamente il costruttore per riferimenti universali di `Person` per quei tipi di argomenti che vogliamo siano gestiti dai costruttori per copia e per spostamento della classe, ma non abbiamo ancora visto come distinguere gli argomenti interi e non-interi. Dopotutto, questo era il nostro primo obiettivo; il problema dell'ambiguità del costruttore è stato semplicemente qualcosa che abbiamo incontrato lungo il percorso.

Tutto ciò che dobbiamo fare (e con questo abbiamo davvero concluso) è (1) aggiungere un costruttore in overloading di `Person` per gestire gli argomenti interi e (2) vincolare ulteriormente il costruttore template-izzato, in modo che sia disabilitato per questi argomenti. Aggiungiamo questi ultimi "ingredienti" al "piatto" che abbiamo cucinato finora, cuociamo il tutto a fuoco lento e godiamoci il profumo del successo:

```
class Person {
public:
  template<
     typename T,
     typename = std::enable_if_t<
       !std::is_base_of<Person, std::decay_t<T>>::value
        &&
       !std::is_integral<std::remove_reference_t<T»::value
     >
>

explicit Person(T&& n)                        // costr. per std::strings
                                              e
: name(std::forward<T>(n))                    // argomenti convertibili
{ ... }                                       // in std::strings
```

```
explicit Person(int idx)                  // costr. per argomenti interi
: name(nameFromIdx(idx))
{ ??? }

…                                         // costr. per copia e spostamento ecc.

private:
    std::string name;
};
```

Voilà! Una vera bellezza! D'accordo... una bellezza per chi ha una passione per la metaprogrammazione a template, ma rimane il fatto che questo approccio non solo risolve tutti i problemi, ma ha anche una certa eleganza. Poiché utilizza il perfect forward, offre la massima efficienza e poiché controlla la combinazione riferimenti universali/overloading piuttosto che proibirla, questa tecnica può essere applicata in tutti quei casi in cui l'overloading è inevitabile, come nel caso dei costruttori.

Compromessi

Le prime tre tecniche considerate in questo Elemento (abbandono dell'overloading, passaggio per `const T&` e passaggio per valore) specificano un tipo per ciascun parametro nella funzione (o nelle funzioni) che verrà richiamata. Le ultime due tecniche, tag dispatch e vincolo dei template, usano il perfect-forward, pertanto non specificano il tipo dei parametri. Questa decisione fondamentale (specificare o non specificare un tipo) ha delle conseguenze.

Come regola generale, il perfect-forward è più efficiente, poiché evita di creare oggetti temporanei con il solo scopo di conformarsi al tipo della dichiarazione del parametro. Nel caso del costruttore di `Person`, il perfect-forward permette a una stringa letterale come `"Nancy"` di essere inoltrata al costruttore di `std::string` all'interno di `Person`, mentre le tecniche che non utilizzano il perfect-forward devono creare un oggetto temporaneo `std::string` a partire

dalla stringa letterale e soddisfare la specifica dei parametri per il costruttore di `Person`.

Ma il perfect-forward ha dei difetti. Uno è che alcuni tipi di argomenti non possono essere inoltrati perfettamente, anche se possono essere passati a funzioni che accettano tipi specifici. L'Elemento 30 esplora questi casi di fallimento del perfect-forward.

Un secondo problema è la comprensibilità dei messaggi d'errore che compaiono quando i client passano argomenti non validi. Supponete, per esempio, che un client che crea un oggetto `Person` passi una stringa letterale costituita da "caratteri" `char16_t` (un tipo introdotto in C++11 per rappresentare caratteri a 16 bit) invece di `char` (ovvero ciò di cui è costituita una `std::string`):

```
Person p(u"Konrad Zuse");          // "Konrad Zuse" è costituita da
                                   // caratteri di tipo const char16_t
```

Con i primi tre approcci esaminati in questo Elemento, i compilatori vedranno che i costruttori disponibili accettano `int` o `std::string` e produrranno un messaggio d'errore più o meno chiaro, che spiega che non esiste alcuna conversione da `char16_t[12]` a `int` o `std::string`.

Con un approccio basato sul perfect-forward, invece, l'array di `const char16_t` viene associato al parametro del costruttore senza alcun problema. Da qui viene inoltrato al costruttore del dato membro `std::string` di `Person` ed è solo a questo punto che viene scoperto il problema di corrispondenza fra il chiamante passato (un array di `const char16_t`) e ciò che è richiesto (ogni tipo accettabile per il costruttore di `std::string`). Il messaggio d'errore risultante è, quanto meno, impressionante. Uno dei compilatori che utilizzo abitualmente produce un messaggio d'errore di ben 160 righe.

In questo esempio, il riferimento universale viene inoltrato una sola volta (dal costruttore di `Person` al costruttore di `std::string`), ma più complesso è il sistema, più è probabile che un riferimento universale venga inoltrato attraverso vari strati di chiamate a funzione prima di arrivare finalmente in un punto che determina se il tipo degli argomenti è accettabile. Più volte il riferimento universale viene inoltrato, più complesso sarà il messaggio d'errore che dice che qualcosa non va. Molti sviluppatori trovano che questo problema, da solo, basta

a riservare i parametri riferimento universale alle interfacce quando le prestazioni sono un fattore importante.

Nel caso di `Person`, sappiamo che il parametro riferimento universale della funzione che esegue l'inoltro si suppone che sia un inizializzatore per una `std::string`; pertanto possiamo utilizzare `static_assert` per verificare che possa giocare tale ruolo. Il type trait `std::is_constructible` svolge un test in fase di compilazione per determinare se un oggetto di un tipo può essere costruito a partire da un oggetto (o da un insieme di oggetti) di un tipo (o di un insieme di tipi) differente, pertanto l'asserzione è facile da scrivere:

```
class Person {
public:
   template<         // come prima
      typename T,
      typename = std::enable_if_t<
        !std::is base of<Person, std::decay t<T>>::value
        &&
        !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)
  : name(std::forward<T>(n))
  {
       // asserisce che una std::string possa essere creata da un
oggetto T
     static_assert(
       std::is_constructible<std::string, T>::value,
       "Parameter n can't be used to construct a std::string"
  );
  …
                      // qui va il solito costruttore
  }
  …                 // resto della classe Person (come prima)
};
```

Questo fa sì che venga prodotto il messaggio d'errore specificato se il codice client tenta di creare una `Person` da un tipo che non può essere utilizzato per costruire una `std ::string`. Sfortunatamente, in questo esempio `static_assert` è nel corpo del costruttore, ma il codice di inoltro, facendo parte dell'elenco di inizializzazione del membro, la precede. Con i compilatori utilizzati da me, il risultato è che il messaggio, questa volta comprensibile, derivante da `static_assert` appare solo *dopo* che i normali messaggi d'errore

(oltre 160 righe di messaggi) sono stati emessi.

Argomenti da ricordare

- Fra le alternative alla combinazione riferimenti universali/overloading vi sono l'uso di nomi distinti per le funzioni, il passaggio di parametri per riferimento lvalue a `const`, il passaggio di parametri per valore e la tecnica tag dispatch.
- Il vincolo dei template tramite `std::enable_if` consente l'uso congiunto di riferimenti universali e overloading, ma controlla le condizioni in cui i compilatori possono utilizzare gli overloading dei riferimenti universali.
- I parametri riferimento universale spesso offrono dei vantaggi in termini di efficienza, ma in genere portano degli svantaggi in termini di usabilità.

Elemento 28 – Il collasso dei riferimenti

Nell'Elemento 23 abbiamo visto che quando a una funzione template viene passato un argomento, il tipo dedotto per il parametro template codifica il fatto che l'argomento sia un lvalue o un rvalue. In tale Elemento non menzioniamo però il fatto che ciò si verifica solo quando l'argomento viene utilizzato per inizializzare un parametro che è un riferimento universale, ma tale omissione ha un buon motivo: i riferimenti universali vengono introdotti solo nell'Elemento 24. Insieme, queste due osservazioni relative ai riferimenti universali e alla codifica lvalue/rvalue significano che per questo template,

```
template<typename T>
void func(T&& param);
```

il parametro template dedotto `T` codifica se l'argomento passato a `param` è un lvalue oppure un rvalue.

Il meccanismo di codifica è semplice. Quando come argomento viene passato un lvalue, `T` viene dedotto come un riferimento lvalue. Quando viene passato un rvalue, `T` viene dedotto come un non-riferimento (notate l'asimmetria: gli lvalue vengono codificati come riferimenti lvalue, mentre gli rvalue vengono codificati come *non-riferimenti*). Pertanto:

```
Widget widgetFactory();     // funzione che restituisce un rvalue

Widget w;                   // una variabile (un lvalue)

func(w);                    // richiama func con un lvalue; T dedotto
                            // come Widget&

func(widgetFactory());      // richiama func con un rvalue; T dedotto
                            // come Widget
```

In entrambe le chiamate a `func`, viene passato un `Widget`, ma poiché un `Widget` è un lvalue e un altro è un rvalue, per il parametro template `T` vengono dedotti tipi differenti. Questo, come vedremo presto, è ciò che determina se i riferimenti universali divengono riferimenti rvalue oppure riferimenti lvalue e anche il meccanismo mediante il quale `std::forward` svolge proprio lavoro.

Prima di poter osservare più da vicino `std::forward` e i riferimenti universali, occorre notare che i riferimenti di riferimenti sono illegali in C++. Se doveste dichiararne uno, il compilatore non ve la farà passare liscia:

```
int x;
…
auto& & rx = x;             // errore! Non si può dichiarare un riferimento di un riferimento
```

Ma considerate ciò che accade quando a un template di funzione che accetta un riferimento universale viene passato un lvalue:

```
template<typename T>
void func(T&& param);                   // come prima

func(w);                                // richiama func con un lvalue;
                                        // T dedotto come Widget&
```

Se prendiamo il tipo dedotto per `T` (per esempio `Widget&`) e lo utilizziamo per

istanziare il template, otteniamo:

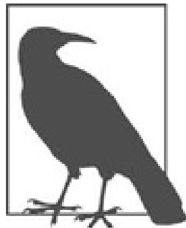
```
void func(Widget& && param);
```

ovvero un riferimento a un riferimento. Tuttavia il compilatore non se ne lamenta. Sappiamo dall'Elemento 24 che poiché il riferimento universale `param` è stato inizializzato con un lvalue, il tipo di `param` si suppone che sia un riferimento a un lvalue, ma come arriva il compilatore al risultato di prendere il tipo dedotto per `T` e sostituirlo nel template come il seguente, che risulta essere la signature finale della funzione?

```
void func(Widget& param);
```

La risposta è legata al *collasso dei riferimenti*. In pratica, *noi non possiamo* dichiarare riferimenti a riferimenti, mentre *i compilatori* possono produrli, in particolari contesti; fra questi, vi è l'istanziazione di template. Quando i compilatori generano riferimenti a riferimenti, ciò che accade successivamente è stabilito dal collasso dei riferimenti.

Vi sono due tipi di riferimenti (lvalue e rvalue), pertanto vi sono quattro possibili combinazioni fra riferimenti (lvalue a lvalue, lvalue a rvalue, rvalue a lvalue e rvalue a rvalue). Se sorge un riferimento di un riferimento in un contesto in cui questo è permesso (ovvero durante l'istanziazione di un template), i riferimenti collassano in un unico riferimento in base alla seguente regola:



Se almeno uno dei due riferimenti è un riferimento lvalue, il risultato è un riferimento lvalue. Altrimenti (ovvero se entrambi sono riferimenti rvalue), il risultato è un riferimento rvalue.

Nell'esempio precedente, la sostituzione del tipo dedotto `Widget&` nel template `func` fornisce un riferimento rvalue a un riferimento lvalue e la regola di collasso dei riferimenti ci dice che il risultato è un riferimento lvalue.

Il collasso dei riferimenti è un elemento fondamentale per il funzionamento di `std ::forward`. Come descritto nell'Elemento 25, `std::forward` si applica ai parametri riferimento universale e dunque un caso d'uso comune ha il seguente aspetto:

```
template<typename T>
```

```
void f(T&& fParam)
{
    …                                       // fa la stessa cosa

    someFunc(std::forward<T>(fParam));      // inoltra fParam
}                                           // a someFunc
```

Poiché `fParam` è un riferimento universale, sappiamo che il parametro tipo `T` codifica il fatto che l'argomento passato a `f` (ovvero l'espressione utilizzata per inizializzare `fParam`) sia un lvalue o un rvalue. Il compito di `std::forward` è quello di convertire `fParam` (un lvalue) in un rvalue se (e solo se) `T` codifica il fatto che l'argomento passato `f` era un rvalue, ovvero se `T` è di un tipo non-riferimento.

Ecco come `std::forward` può essere implementata per fare proprio questo:

```
template<typename T>                        // nel
T&& forward(typename                        // namespace
      remove_reference<T>::type& param)     // std
{
    return static_cast<T&&>(param);
}
```

Non è esattamente conforme allo standard (ho omesso alcuni dettagli dell'interfaccia), ma le differenze sono irrilevanti per gli scopi della discussione, ovvero di comprendere come si comporta `std::forward`.

Supponete che l'argomento passato a `f` sia un lvalue di tipo `Widget`. `T` verrà dedotto come `Widget&` e la chiamata a `std::forward` lo istanzierà come `std::forward<Widget&>`. Inserendo `Widget&` nell'implementazione di `std::forward` si ottiene:

```
WidgetS && forward(typename
         remove_reference<Widget&>::type& param) {
return static_cast<Widget& &&>(param); }
```

Il type trait `std::remove_reference<Widget&>::type` fornisce `Widget` (vedi Elemento 9), pertanto `std::forward` diviene:

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

Il collasso dei riferimenti viene applicato anche al tipo restituito e alla conversione, e il risultato è la versione finale di `std::forward` per la chiamata:

```
Widget& forward(Widget& param)           // sempre nel
{ return static_cast<WidgetS>(param); }  // namespace std
```

Come potete vedere, quando a un template di funzione `f` viene passato un argomento lvalue, `std::forward` viene istanziato per accettare e restituire un riferimento lvalue. La conversione all'interno di `std::forward` non fa nulla, poiché il tipo di `param` è già `Widget&`, pertanto la sua conversione in `Widget&` non ha alcun effetto. Un argomento lvalue passato a `std::forward` restituirà pertanto un riferimento lvalue. Per definizione, i riferimenti lvalue sono lvalue e pertanto il passaggio di un lvalue a `std::forward` fa sì che venga restituito un rvalue, proprio come deve essere.

Ora supponete che l'argomento passato a `f` sia un rvalue di tipo `Widget`. In questo caso, il tipo dedotto per `T`, il parametro del tipo di `f`, sarà semplicemente `Widget`. La chiamata all'interno di `f` a `std::forward` sarà pertanto `std::forward<Widget>`. Sostituendo `Widget` a `T` nell'implementazione di `std::forward` si ottiene:

```
Widget&& forward(typename
         remove_reference<Widget>::type& param)
{ return static_cast<Widget&&>(param); }
```

Applicando `std::remove_reference` al tipo non-riferimento `Widget` si ottiene lo stesso tipo di partenza (`widget`), pertanto `std::forward` diviene:

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

Qui non vi sono riferimenti a riferimenti, quindi non si verifica alcun collasso fra riferimenti e questa è la versione istanziata finale di `std::forward` per la chiamata.

I riferimenti rvalue restituiti dalle funzioni sono definiti per essere rvalue, pertanto, in questo caso, `std::forward` trasformerà il parametro `fParam` di `f` (un

lvalue) in un rvalue. Il risultato finale è che un argomento rvalue passato a `f` verrà inoltrato a `someFunc` come un rvalue, che è esattamente ciò che deve accadere.

In C++14, l'esistenza di `std::remove_reference_t` consente di implementare `std ::forward` in modo un po' più compatto:

```
template<typename T>                          // C++14; sempre nel
T&& forward(remove_reference_t<T>& param)     // namespace std
{
   return static_cast<T&&>(param);
}
```

Il collasso dei riferimenti si verifica in quattro contesti. Il primo, più comune, è l'istanziazione di template. Il secondo è la generazione del tipo per le variabili `auto`. I dettagli sono sostanzialmente gli stessi per i template, poiché la deduzione del tipo per le variabili `auto` coincide essenzialmente con la deduzione del tipo per i template (vedi Elemento 2). Considerate ancora una volta quest'esempio presentato in precedenza nel nell'Elemento:

```
template<typename T>
void func(T&& param);

Widget widgetFactory();      // funzione che restituisce un rvalue

Widget w;                    // una variabile (un lvalue)

func(w);                     // richiama func con un lvalue; T dedotto
                             // come Widget&

func(widgetFactory());       // richiama func con un rvalue; T dedotto
                             // come Widget
```

Questo può essere simulato con un `auto.` La dichiarazione

```
auto&& w1 = w;
```

inizializza `w1` con un lvalue, deducendo pertanto il tipo `Widget&` per `auto.` Inserendo `Widget&` per `auto` nella dichiarazione per `w1` si ottiene questo codice, che rappresenta un riferimento di riferimento,

```
WidgetS SS w1 = w;
```

che, dopo il collasso dei riferimenti, diviene

```
WidgetS w1 = w;
```

Come risultato, `w1` è un riferimento lvalue.

Al contrario, questa dichiarazione,

```
auto&& w2 = widgetFactory();
```

inizializza `w2` con un rvalue, facendo sì che il tipo non-riferimento `Widget` venga dedotto per `auto.` Sostituendo `Widget` ad `auto` si ottiene:

```
WidgetSS w2 = widgetFactory();
```

Qui non vi sono riferimenti a riferimenti e dunque siamo cavallo: `w2` è un riferimento rvalue.

Ora abbiamo finalmente la possibilità di comprendere appieno i riferimenti universali introdotti nell'Elemento 24. Un riferimento universale non è un nuovo tipo di riferimento, in realtà è un riferimento rvalue in un contesto in cui sono soddisfatte le seguenti due condizioni.

- **La deduzione del tipo distingue lvalue e rvalue.** Gli lvalue di tipo `T` vengono dedotti in modo che abbiano tipo `T&,` mentre gli rvalue di tipo `T` forniscono `T` come tipo dedotto;
- **Si verifica il collasso dei riferimenti.**

Il concetto dei riferimenti universali è utile, poiché ci evita di dover riconoscere l'esistenza dei contesti di collasso dei riferimenti, di dedurre mentalmente tipi differenti per lvalue e rvalue e di applicare la regola di collasso dei riferimenti dopo aver sostituito mentalmente i tipi dedotti nei contesti in cui si presentano.

Abbiamo detto che vi sono quattro contesti di questo tipo, ma ne abbiamo introdotti solo due: l'istanziazione di template e la generazione del tipo con `auto`. Il terzo è la generazione e l'uso di `typedef` e delle dichiarazioni alias (vedi

Elemento 9). Se, durante la creazione o valutazione di un `typedef`, sorgono dei riferimenti a riferimenti, per eliminarli interviene il collasso dei riferimenti. Per esempio, supponete di avere una classe template `Widget` contenente un `typedef` per un tipo riferimento rvalue,

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
  …
};
```

e supponete di istanziare Widget con un tipo riferimento lvalue:

```
Widget<int&> w;
```

Sostituendo `int&` per `T` nel template di `Widget`, si ottiene il seguente `typedef`:

```
typedef int& && RvalueRefToT;
```

Il collasso dei riferimenti lo riduce a questo,

```
typedef int& RvalueRefToT;
```

che chiarisce il fatto che il nome scelto per il `typedef` forse non è descrittivo quanto avremmo sperato: *RvalueRef*`ToT` è un `typedef` per un *riferimento lvalue* quando Widget viene istanziato con un tipo riferimento lvalue.

L'ultimo contesto in cui si svolge il collasso dei riferimenti è legato agli usi di `decltype`. Se, durante l'analisi di un tipo che richiama `decltype`, sorge un riferimento di riferimento, interverrà il collasso dei riferimenti per eliminarlo (per informazioni su `decltype`, consultate l'Elemento 3).

Argomenti da ricordare

- Il collasso dei riferimenti si verifica in quattro contesti: istanziazione di template, generazione del tipo con auto, creazione e uso di `typedef` e dichiarazioni alias e `decltype`.
- Quando i compilatori generano un riferimento a un riferimento in un contesto di collasso dei riferimenti, il risultato diviene un singolo riferimento. Se almeno uno dei riferimenti originali è un lvalue, il risultato è un riferimento lvalue. Altrimenti sarà un riferimento rvalue.
- I riferimenti universali sono riferimenti lvalue nei contesti in cui la

deduzione del tipo distingue gli lvalue dagli rvalue e dove si verifica il collasso dei riferimenti.

Elemento 29 – Operazioni di spostamento: se non sono disponibili, economiche o utilizzate

Le semantiche di spostamento sono senza dubbio *la* funzionalità principale del C++11. Si dice che "Lo spostamento di container, oggi, è altrettanto economico della copia di puntatori!" e "La copia di oggetti temporanei è oggi così efficiente che programmare per evitarla diventa una sorta di ostacolo all'ottimizzazione!". Tali affermazioni sono facili da comprendere. Le semantiche di spostamento rappresentano davvero una funzionalità importante. Non solo consentono ai compilatori di sostituire le costose operazioni di copia con i, molto più economici, spostamenti, ma, in realtà *richiedono* che sia (quasi) sempre così. Diciamo "quasi", perché devono essere soddisfatte le condizioni corrette. Prendete il vostro codice C++98, ricompilatelo con un compilatore C++11 e la Libreria Standard e, *d'un tratto*, il vostro software sarà più veloce.

Le semantiche di spostamento possono davvero fare la differenza e ciò conferisce a questa funzionalità un'aura di leggenda. Tuttavia, le leggende, di solito, sono il risultato di un'esagerazione. Scopo di questo Elemento è quello di aiutarci a rimettere i piedi per terra.

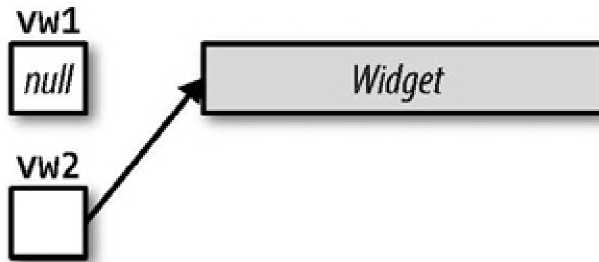
Iniziamo osservando che molti tipi non supportano la semantica di spostamento. L'intera Libreria Standard C++98 è stata rielaborata per il C++11, in modo da aggiungere le operazioni di spostamento per i tipi in cui lo spostamento potesse essere implementato in modo più rapido rispetto alla copia; anche l'implementazione dei componenti della libreria è stata riveduta per sfruttare queste operazioni. Tuttavia, è probabile che abbiate a disposizione una base di codice che non è stata completamente aggiornata per sfruttare le nuove funzionalità del C++11. Per i tipi (dell'applicazione o delle librerie che utilizzate) per i quali non sono state eseguite modifiche da parte del C++11, il nuovo supporto per gli spostamenti nei compilatori non farà una grande differenza. Certamente il C++11 intenderà generare delle operazioni di spostamento per le classi che non le offrono, ma ciò solo per le classi che non dichiarano operazioni di copia, di spostamento o distruttori (vedi Elemento 17). I

dati membro o le classi base dei tipi per i quali è disabilitato lo spostamento (per esempio tramite una cancellazione, vedi Elemento 11) sopprimeranno sempre le operazioni di spostamento generate dal compilatore. Per i tipi che non offrono un supporto esplicito per lo spostamento e che non si qualificano per le operazioni di spostamento generate dal compilatore, non vi è alcun motivo di aspettarsi che il C++11 fornisca un qualche miglioramento prestazionale rispetto al C++98.

Perfino i tipi che offrono un supporto esplicito per le operazioni di spostamento potrebbero non conseguire i vantaggi desiderati. Per esempio, tutti i container nella Libreria Standard C++11 supportano lo spostamento, ma sarebbe un errore presumere che lo spostamento di tutti i container sia necessariamente economico. Per alcuni container, il motivo è che non esiste un modo davvero economico per spostarne il contenuto. Per altri, le operazioni di spostamento davvero economiche dei container hanno dei dettagli che gli elementi del container non riescono a soddisfare.

Considerate `std::array`, un nuovo container C++11. `std::array` è sostanzialmente un array standard con un'interfaccia STL. Si tratta di una struttura fondamentalmente differente dagli altri container standard, ognuno dei quali memorizza il proprio contenuto nello heap. Gli oggetti di questi tipi di container contengono (come dati membro), teoricamente, solo un puntatore alla memoria dello heap che conserva il contenuto del container (la realtà è più complessa, ma per gli scopi di questa analisi, tali differenze sono ininfluenti). L'esistenza di questo puntatore consente di spostare il contenuto di un intero container in un tempo costante: basta copiare il puntatore dal container di origine a quello di destinazione e impostare il puntatore di origine a null:

```
std::vector<Widget> vw1;

// inserisce i dati in vw1

…

// sposta vw1 in vw2. Opera in
// un tempo costante. Solo i puntatori
// in vw1 e vw2 vengono modificati
auto vw2 = std::move(vw1);
```



Gli oggetti `std::array` non hanno questo puntatore, poiché i dati del loro contenuto sono conservati direttamente nell'oggetto `std::array`:

```
std::array<Widget, 10000> aw1;

// inserisce i dati in aw1

…

// sposta aw1 in aw2. Opera in
// un tempo lineare. Tutti gli elementi
// di aw1 vengono spostati in aw2
auto aw2 = std::move(aw1);
```

Notate che gli elementi in `aw1` vengono *spostati* in `aw2`. Supponendo che `Widget` sia un tipo in cui lo spostamento è più veloce rispetto alla copia, lo spostamento

di uno `std::array` di `Widget` sarà più veloce rispetto alla copia. Pertanto, `std::array` offre certamente il supporto dello spostamento. Tuttavia, sia lo spostamento sia la copia di uno `std::array` hanno una complessità computazionale di tipo lineare, poiché deve essere copiato o spostato ciascun elemento del container. Questo non va d'accordo con l'affermazione "Oggi lo spostamento di un container è economico quanto assegnare un paio di puntatori" che si sente dire talvolta.

D'altra parte, std::string offre anche delle operazioni di spostamento in tempo costante e di copia in tempo lineare. Ciò fa pensare che lo spostamento sia più veloce rispetto alla copia, ma non sempre le cose stanno così. Molte implementazioni di stringhe impiegano l'*ottimizzazione SSO* (Small String Optimization), in base alla quale, le stringhe "piccole" (per esempio quelle con una capacità non superiore a 15 caratteri) vengono conservate in un buffer interno dell'oggetto std::string; non viene utilizzata memoria tratta dallo heap. Lo spostamento di piccole stringhe utilizzando un'implementazione basata su SSO non è più veloce rispetto alla copia, poiché non è applicabile il trucco di copiare solo un puntatore, su cui si basa generalmente il vantaggio prestazionale degli spostamenti rispetto alle copie.

La motivazione per l'ottimizzazione SSO è il fatto che le stringhe brevi sono la norma per molte applicazioni. L'utilizzo di un buffer interno per memorizzare il contenuto di tali stringhe elimina la necessità di allocare dinamicamente della memoria per loro, e questo, in genere, si traduce in un vantaggio in termini di efficienza. Un'implicazione di questo vantaggio, tuttavia, è il fatto che gli spostamenti non sono più veloci delle copie, anche se si può adottare un approccio a "bicchiere mezzo pieno" e dire che per queste stringhe, la copia non è più lenta rispetto allo spostamento.

Anche per i tipi che supportano le operazioni di spostamento accelerate, alcune situazioni di spostamento apparentemente "sicure" possono trasformarsi in altrettante copie. L'Elemento 14 spiega che alcune operazioni per container nella Libreria Standard offrono la garanzia forte per la sicurezza delle eccezioni; e per garantire che il codice C++98 preesistente che dipende da tale garanzia continui a funzionare nel passaggio al C++11, le operazioni di copia sottostanti possono essere sostituite da altrettante operazioni di spostamento solo se si sa che non vengono lanciate le operazioni di spostamento. Una conseguenza è che anche se un tipo offre delle operazioni di spostamento che sono più efficienti rispetto alle corrispondenti operazioni di copia e anche se, in un determinato punto del

codice, un’operazione di spostamento sarebbe generalmente appropriata (per esempio se l’oggetto di origine è un rvalue), i compilatori possono comunque essere costretti a richiamare un’operazione di copia, poiché la corrispondente operazione di spostamento non è stata dichiarata `noexcept`.

Vi sono pertanto varie situazioni in cui la semantica di spostamento del C++11 non offre particolari vantaggi.

- **Nessuna operazione di spostamento:** l’oggetto da spostare non offre operazioni di spostamento. La richiesta di spostamento diviene pertanto una richiesta di copia.
- **Spostamento non veloce:** l’oggetto da spostare ha delle operazioni di spostamento, ma non sono più veloci rispetto alle operazioni di copia.
- **Spostamento non utilizzabile:** il contesto in cui si svolgerebbe lo spostamento richiede un’operazione di spostamento che non ammette eccezioni, ma tale operazione non è dichiarata `noexcept`.

Vale anche la pena di menzionare un’altra situazione in cui le semantiche di spostamento non offrono alcun vantaggio in termini di efficienza.

- **L’oggetto di origine è un lvalue:** con pochissime eccezioni (vedi, per esempio, l’Elemento 25) solo gli rvalue possono essere utilizzati come origine di un’operazione di spostamento.

Ma il titolo di questo Elemento è di *supporre* che le operazioni di spostamento non siano presenti, non siano economiche e non vengano utilizzate. Questo è tipicamente il caso del codice generico, ovvero dei template, poiché non si conoscono tutti i tipi con cui si sta lavorando. In tali circostanze, in C++98 (prima che esistesse la semantica di spostamento) occorreva essere conservativi sulla copia degli oggetti. È anche il caso del codice “instabile”, ovvero codice in cui le caratteristiche dei tipi utilizzati sono soggette a modifiche relativamente frequenti.

Spesso, tuttavia, si conoscono i tipi utilizzati dal codice e si può contare sul fatto che le loro caratteristiche non cambino (in termini di supporto o meno di operazioni di spostamento economiche). In questo caso, basta cercare dettagli sul supporto dello spostamento per i tipi utilizzati. Se tali tipi offrono operazioni di spostamento economiche e se si utilizzano questi oggetti nei contesti in cui tali operazioni di spostamento verranno richiamate, si può contare con sicurezza sul

fatto che le semantiche di spostamento sostituiscano le operazioni di copia, con i relativi vantaggi prestazionali.

Argomenti da ricordare

- Supporre che le operazioni di spostamento non siano presenti, non siano economiche e non vengano utilizzate.
- Nel codice con tipi noti o con il supporto delle semantiche di spostamento, non è necessario fare supposizioni.

Elemento 30 – Casi problematici di perfect-forward

Una delle funzionalità più blasonate del C++11 e il perfect-forward. Se si chiama “perfetto”, sarà, per forza di cose... *perfetto*! Purtroppo, mettendoci mano, si scopre che c’è una certa differenza fra un perfetto ideale e un perfetto reale. Il perfect-forward del C++11 è un’ottima funzionalità, ma raggiunge la vera perfezione solo se si tengono in considerazione alcuni piccoli dettagli. Questo Elemento è dedicato proprio a tali dettagli.

Prima di affrontare la nostra esplorazione, vale la pena di ripassare che cosa si intende con “perfect-forward”. Con “forward”, inoltro, si intende semplicemente che una funzione passa, *inoltra*, i propri parametri a un’altra funzione. L’obiettivo è che la seconda funzione (quella che riceve l’inoltro) riceva gli stessi oggetti che la prima funzione (quella che esegue l’inoltro) ha ricevuto. Questo taglia fuori i parametri passati per valore, poiché sono solo *copie* di ciò che ha passato il chiamante originale. Vogliamo che la funzione che riceve l’inoltro sia in grado di operare sugli oggetti originariamente passati. Si devono scartare anche i parametri puntatore, poiché non vogliamo costringere i chiamanti a passare dei puntatori. Parlando di inoltro, in generale, consideriamo parametri che sono riferimenti.

Il perfect-forward significa che non inoltriamo semplicemente degli oggetti, inoltriamo anche le loro caratteristiche salienti: il loro tipo, il fatto che si tratti di valori lvalue o rvalue, e il fatto che siano `const` o `volatile`. Unendo

l’osservazione precedente (che abbiamo a che fare con parametri riferimento), ciò significa che stiamo utilizzando riferimenti universali (Elemento 24), poiché solo i riferimenti universali codificano informazioni relative al fatto che gli argomenti passati siano lvalue o rvalue.

Supponiamo di avere una funzione `f` e di voler scrivere una funzione (in realtà un template di funzione) che esegua l’inoltro a tale funzione. Sostanzialmente il tutto ha il seguente aspetto:

```
template<typename T>
void fwd(T&& param)                    // accetta qualsiasi argomento
{
   f(std::forward<T>(param));          // lo inoltra a f
}
```

Le funzioni che eseguono l’inoltro sono, per definizione, generiche. Il template `fwd`, per esempio, accetta ogni tipo di argomento e inoltra ciò che riceve. Un’estensione logica di questa genericità è che le funzioni che eseguono l’inoltro non siano semplici template, ma template *variadic*, che dunque accettino qualsiasi numero di argomenti. La forma variadic di `fwd` ha il seguente aspetto:

```
template<typename... Ts>
void fwd(Ts&&... params)               // accetta qualsiasi numero di argomenti
{
   f(std::forward<Ts>(params)...);     // li inoltra tutti a f
}
```

Questa è la forma che vedrete, fra l’altro, nelle funzioni di *emplacement* dei container standard (Elemento 42) e che avete visto nelle funzioni *factory* dei puntatori smart, `std::make_shared` e `std::make_unique` (Elemento 21).

Data la nostra funzione obiettivo `f` e la nostra funzione di inoltro fwd, il perfect-forward *non ha successo* se accade che chiamando `f` con un determinato argomento si ottiene una cosa, mentre chiamando `fwd` con lo stesso argomento si ottiene qualcosa di differente:

```
f( expression );     // se qui si ottiene una cosa,
fwd( expression );   // e qui si ottiene qualcosa di differente,
                     // fwd non inoltra perfettamente expression a
                     f
```

Vari tipi di argomenti provocano questo tipo di problema. Sapere quali sono e come aggirare il problema è importante; pertanto, vediamo innanzitutto quali argomenti non sono adatti al perfect-forward.

Inizializzatori a graffe

Supponete che `f` sia dichiarata nel seguente modo:

```
void f(const std::vector<int>& v);
```

In tal caso, richiamando `f` con un inizializzatore a graffe, il codice viene compilato,

```
f({ 1, 2, 3 });      // OK, "{1, 2, 3}" implicitamente
                     // convertito in std::vector<int>
```

mentre passando lo stesso inizializzatore a graffe a `fwd` il codice non viene compilato:

```
fwd({ 1, 2, 3 }); // errore! Non viene compilato
```

Questo perché l'uso di un inizializzatore a graffe è uno dei casi in cui il perfect-forward non si può applicare.

Tutti i casi di fallimento di questo tipo hanno la stessa causa. In una chiamata diretta a `f` (come `f({ 1, 2, 3 })`), i compilatori vedono gli argomenti passati nel punto di chiamata e vedono anche i tipi dei parametri dichiarati da `f`. Confrontano gli argomenti nel punto di chiamata con le dichiarazioni dei parametri, ne controllano la compatibilità e, se necessario, svolgono le opportune conversioni implicite, per far sì che la chiamata abbia successo. Nell'esempio precedente, generano da `{ 1, 2, 3 }` un oggetto `std::vector<int>` temporaneo, in modo che il parametro `v` di `f` abbia un oggetto `std::vector<int>` cui associarsi.

Chiamando `f` indirettamente tramite il template della funzione di inoltro `fwd`, i

compilatori non potranno più confrontare gli argomenti passati nel punto di chiamata di `fwd` con le dichiarazioni dei parametri in `f`. Al contrario, *dedurranno* i tipi degli argomenti passati a `fwd` e confronteranno i tipi dedotti con le dichiarazioni dei parametri di `f`. Il perfect-forward non funziona quando si verifica uno dei due casi seguenti.

- **I compilatori non sono in grado di dedurre un tipo** da uno o più dei parametri di `fwd`. In tal caso, il codice non verrà compilato.
- **I compilatori deducono il tipo "errato"** per uno o più dei parametri di `fwd`. Qui "errato" può significare che l'istanziazione di `fwd` non può essere compilata con i tipi che sono stati dedotti, ma può anche significare che la chiamata a `f` che utilizza i tipi dedotti per `fwd` si comporta in modo differente rispetto a una chiamata diretta a `f` con gli argomenti effettivamente passati a `fwd`. Un'origine di questo comportamento divergente può essere il caso di una `f` che è una funzione in overloading: a causa di una deduzione "errata" dei tipi, l'overload di `f` richiamata all'interno di `fwd` può essere differente dall'overload che verrebbe richiamata se `f` fosse stata richiamata direttamente.

Nella chiamata `fwd({ 1, 2, 3 })` precedente, il problema è che passando un inizializzatore a graffe al parametro template di una funzione che non è dichiarata come `std::initializer_list`, si dichiara, come dice lo standard, un "contesto non-dedotto". Ciò significa che i compilatori non devono poter dedurre un tipo per l'espressione `{ 1, 2, 3 }` nella chiamata `fwd`, poiché il parametro di `fwd` non è dichiarato come una `std::initializer_list`. Non potendo dedurre un tipo per il parametro di `fwd`, i compilatori devono, comprensibilmente, rifiutare la chiamata.

Un aspetto interessante è che l'Elemento 2 spiega che la deduzione del tipo ha successo per le variabili `auto` inizializzate con un inizializzatore a graffe. Tali variabili sono considerate oggetti `std::initializer_list` e questo ci dà una semplice soluzione per i casi in cui il tipo che la funzione di inoltro dovrebbe dedurre è una `std::initializer_list`: basta dichiarare una variabile locale utilizzando `auto` e poi passare la variabile locale alla funzione che esegue l'inoltro.

```
auto il = { 1, 2, 3 };                    // il tipo di il è dedotto
                                          come
                                          // std::initializer
```

```
                                        list<int>

fwd(il);                                // PK, perfect-forward di
                                        il a f
```

0 o Null come puntatori nulli

L'Elemento 8 spiega che quando si tenta di passare `0` o **NULL** come puntatori nulli a un template, la deduzione del tipo è disorientata, deducendo per l'argomento passato un tipo intero (tipicamente `int`) invece di un tipo puntatore. Il risultato è che né `0` né `NULL` possono essere inoltrati "perfettamente" come puntatori `null`. La soluzione è semplice: passare `nullptr` invece di `0 o NULL`. Per i dettagli, consultate l'Elemento 8.

Dati membro static const interi solo nella dichiarazione

Come regola generale, non vi è alcuna necessità di definire i dati membro `static const` interi all'interno delle classi; bastano le dichiarazioni. Questo perché i compilatori eseguono la *propagazione const* sul valore di tali membri, eliminando la necessità di riservare ulteriore memoria. Per esempio, considerate il seguente codice:

```
class Widget {
public:
   static const std::size_t MinVals   // dichiarazione di MinVals
   = 28;
   …
};
...                                   // nessuna definizione per
                                      MinVals

std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals); // uso di MinVals
```

Qui utilizziamo `Widget::MinVals` (di conseguenza, semplicemente `MinVals`) per specificare la capacità iniziale di `widgetData`, anche se `MinVals` non ha una

propria definizione. I compilatori aggirano la definizione mancante (e sono obbligati a farlo) inserendo il valore 28 in tutti punti in cui è menzionato `MinVals`. Il fatto che non sia stata riservata memoria per il valore `MinVals` non è un problema. Se servisse l'indirizzo di `MinVals` (per esempio se qualcuno creasse un puntatore a `MinVals`), allora `MinVals` richiederebbe della memoria (il puntatore deve avere qualcosa a cui puntare) e il codice precedente, anche se venisse compilato, non passerebbe la fase di linking finché non fosse fornita una definizione per `MinVals`.

Detto questo, immaginate che `f` (la funzione cui `fwd` inoltra i propri argomenti) sia dichiarata nel seguente modo:

```
void f(std::size_t val);
```

Richiamare `f` con `MinVals` va bene, poiché i compilatori non faranno altro che sostituire `MinVals` con il suo valore:

```
f(Widget::MinVals);               // OK, trattato come "f(28)"
```

Purtroppo, le cose non vanno altrettanto bene se tentiamo di richiamare `f` attraverso `fwd`:

```
fwd(Widget::MinVals);             // errore! Non passa il linking
```

Questo codice verrà compilato, ma non dovrebbe riuscire a passare la fase di linking. Se questo vi ricorda ciò che accade se scriviamo del codice che prende l'indirizzo di `MinVals`, è proprio così, poiché il problema è sostanzialmente lo stesso.

Anche se nulla nel codice sorgente prende l'indirizzo di `MinVals`, il parametro di `fwd` è un riferimento universale; e i riferimenti, nel codice generato dai compilatori, vengono solitamente trattati come puntatori. Nel codice binario prodotto per il programma (e nell'hardware), i puntatori e i riferimenti sono sostanzialmente la stessa cosa. A questo livello, è proprio vero che i riferimenti non sono altro che puntatori, deindirizzati automaticamente. Poiché le cose stanno così, il passaggio di `MinVals` per riferimento è sostanzialmente la stessa cosa del passaggio per puntatore e, per questo motivo, vi deve essere della memoria cui il puntatore può puntare. Il passaggio di dati membri `static const` interi per riferimento, quindi, generalmente richiede che essi siano definiti e

questo requisito può far sì che il codice che utilizza il perfect-forward non funzioni laddove invece funziona il codice equivalente senza perfect-forward.

Forse avrete notato le tante "sfumature" che ho impiegato nelle pagine precedenti. Il codice "non *dovrebbe* riuscire a passare la fase di linking". I riferimenti "vengono *solitamente* trattati come puntatori". "Il passaggio di dati membri `static const` interi per riferimento, quindi, *generalmente* richiede che essi siano definiti". È come se io sapessi qualcosa che, in realtà, non voglio dire...

Le cose stanno proprio così. Secondo lo Standard, il passaggio di `MinVals` per riferimento richiede che questo sia definito. Ma non tutte le implementazioni impongono questo requisito. Pertanto, a seconda del compilatore e dei linker impiegati, potreste scoprire di poter inoltrare perfettamente i dati membro `static const` interi anche se non sono stati definiti. Se potete farlo, congratulazioni, ma non aspettatevi che tale codice sia portabile. Per renderlo portabile, fornite semplicemente una definizione per il dato membro `static const` intero in questione. Per `MinVals`, avrà il seguente aspetto:

```
const std::size_t Widget::MinVals;     // nel file cpp di Widget
```

Notate che la definizione non ripete l'inizializzatore (28, nel caso di `MinVals`). Non trascurate questo dettaglio. Se lo dimenticate e fornite l'inizializzatore in entrambe le posizioni, il compilatore se ne lamenterà, ricordandovi di specificarlo una sola volta.

Nomi di funzioni overloaded e nomi template

Supponete che la nostra funzione `f` (quella cui vogliamo inoltrare gli argomenti tramite `fwd`) possa offrire un comportamento personalizzato, ricevendo una funzione che svolga una parte del suo lavoro. Supponendo che questa funzione accetti e restituisca valori `int`, `f` potrebbe essere dichiarata nel seguente modo:

```
void f(int (*pf)(int));     // pf = "processing function"
```

Vale la pena notare che `f` potrebbe anche essere dichiarata utilizzando una sintassi più semplice, senza puntatori. Tale dichiarazione avrebbe il seguente aspetto, e avrebbe lo stesso significato della dichiarazione precedente:

```
void f( int pf(int) );    // dichiara f come sopra
```

In ogni caso, supponete ora di avere una funzione overloaded, `processVal`:

```
int processVal(int value);
int processVal(int value, int priority);
```

Possiamo passare `processVal` a `f`,

```
f(processVal);        // OK
```

ma la cosa è piuttosto sorprendente. `f` richiede come argomento un puntatore a una funzione, ma `processVal` non è un puntatore a funzione e neppure una funzione; è il nome di due diverse funzioni. Tuttavia, i compilatori sanno di quale `processVal` hanno bisogno: quella che corrisponde al tipo di parametro di `f`. Pertanto, scelgono il `processVal` che prende un `int` e passano a `f` l'indirizzo di tale funzione.

Ciò che fa funzionare il tutto è che la dichiarazione di `f` consente ai compilatori di immaginare quale versione di `processVal` è richiesta. Al contrario, `fwd`, essendo un template di funzione, non ha alcuna informazione sul tipo di cui ha bisogno e ciò rende impossibile per i compilatori determinare quale overload dovrebbe essere scelto e passato:

```
fwd(processVal);        // errore! Quale processVal?
```

`processVal` non ha alcun tipo. Senza un tipo, non vi può essere alcuna deduzione del tipo e senza deduzione del tipo, abbiamo un altro caso di fallimento del perfect-forward.

Lo stesso problema sorge se tentiamo di utilizzare un template di funzione invece di (o in aggiunta a) un nome di funzione overloaded. Un template di funzione non rappresenta una funzione, ma *molteplici* funzioni:

```
template<typename T>
T workOnVal(T param)    // template per l'elaborazione dei valori
{ ... }

fwd(workOnVal);         // errore! Quale istanziazione
                        // di workOnVal?
```

Il modo per far sì che una funzione perfect-forward come `fwd` accetti un nome di

funzione overloaded o un nome di template consiste nello specificare manualmente l'overload o l'istanziazione che si desidera venga inoltrata. Per esempio, potete creare un puntatore a funzione dello stesso tipo del parametro di `f`, inizializzare tale puntatore con `processVal` o `workOnVal` (facendo sì che venga selezionata la versione corretta di `processVal` o che venga generata l'istanziazione corretta di `workOnVal`) e passare il puntatore a `fwd`:

```
using ProcessFuncType =                         // crea il typedef;
   int (*)(int);                                // vedi Elemento 9

ProcessFuncType processValPtr = processVal;     // specifica la
                                                // signature per
                                                // processVal

fwd(processValPtr);                             // OK

fwd(static_cast<ProcessFuncType>
(workOnVal));                                   // OK
```

Naturalmente, questo richiede che sappiate il tipo del puntatore a funzione cui `fwd` sta eseguendo l'inoltro. Non è irragionevole presumere che una funzione perfect-forward lo specificherà. Dopotutto, le funzioni perfect-forward sono progettate per accettare qualsiasi cosa, pertanto se non vi è alcuna documentazione che specifica cosa passare, in quale modo è possibile saperlo?

Campi bit

L'ultimo caso di fallimento del perfect-forward si ha quando come argomento di una funzione viene utilizzato un *bitfield*, un campo di bit. Per vedere che cosa significhi questo in pratica, osservate una struttura IPv4 realizzata nel seguente modo:[13]

```
struct IPv4Header {
  std::uint32_t version:4,
           IHL:4,
           DSCP:6,
           ECN:2,
           totalLength:16;
```

```
  ...
};
```

Se la nostra solita funzione `f` (il costante obiettivo della nostra funzione di inoltro `fwd`) è dichiarata in modo da accettare un parametro `std::size_t`, richiamando, per esempio, il campo `totalLength` di un oggetto `IPv4Header`, verrà compilata senza problemi:

```
void f(std::size t sz);                  // funzione da richiamare

IPv4Header h;
...
f(h.totalLength);                        // OK
```

Tentando invece di inoltrare `h.totalLength` a `f` tramite `fwd`, si ottiene tutta un'altra situazione:

```
fwd(h.totalLength); // errore!
```

Il problema è che il parametro di `fwd` è un riferimento e `h.totalLength` è un bitfield non**`-const`.** La cosa può non sembrare così grave, ma lo Standard C++ condanna la combinazione con una descrizione insolitamente chiara: "Un riferimento non**`-const`** non può essere associato a un bitfield". Esiste un ottimo motivo per questa proibizione. I campi di bit possono essere costituiti da parti arbitrarie di word-macchina (per esempio i bit da 3 a 5 di un `int` a 32 bit), ma non vi è alcun modo per accedere direttamente a questi elementi. Ho detto in precedenza che i riferimenti e i puntatori sono la stessa cosa, a livello hardware, e così come non vi è modo di creare un puntatore arbitrario (il C++ stabilisce che la più piccola "cosa" cui si può puntare è un `char`), non vi è alcun modo per associare un riferimento a un singolo bit.

Aggirare l'impossibilità di un perfect-forward di un campo bit è semplice, una volta che si comprende che ogni funzione che accetta un campo bit come argomento riceverà una *copia* del valore del campo bit. Dopotutto, nessuna funzione può legare un riferimento a un campo bit, né una funzione può accettare puntatori a campi bit, poiché i puntatori a campi bit non esistono. L'unico tipo di parametro cui un campo bit può essere passato è costituito dai parametri per valore e, cosa molto interessante, i riferimenti a `const`. Nel caso

dei parametri passati per valore, la funzione richiamata riceve ovviamente una copia del valore del campo bit e ne consegue che, nel caso di un parametro riferimento a `const`, lo Standard richieda che il riferimento si associ effettivamente a una *copia* del valore del campo bit conservata in un oggetto di un tipo intero standard (per esempio `int`). I riferimenti a `const` non si associano ai campi bit, ma a oggetti "normali" all'interno dei quali siano stati copiati i campi bit.

La chiave per passare un campo bit a una funzione perfect-forward, quindi, è quella di sfruttare il fatto che la funzione che riceve l'inoltro riceverà sempre una copia del valore del campo bit. Si può pertanto eseguire una copia del campo e richiamare la funzione che esegue l'inoltro utilizzando tale copia. Nel caso del nostro esempio con `lPv4Header`, ecco il codice che esegue l'operazione:

```
// copia il valore del bitfield; vedi Elemento 6
auto length = static_cast<std::uint16 t>(h.totalLength);

fwd(length);                                // inoltra la copia
…
```

Conclusioni

Nella maggior parte dei casi, il perfect-forward funziona esattamente come previsto. Raramente ci si deve preoccupare troppo. Ma quando non funziona (quando del codice apparentemente ragionevole non viene compilato o, peggio, viene sì compilato, ma non si comporta nel modo previsto), è importante conoscere le "imperfezioni" del perfect-forward. Altrettanto importante è sapere come aggirarle. Nella maggior parte dei casi, non è difficile.

Argomenti da ricordare

- Il perfect-forward non funziona quando non può essere applicata la deduzione del tipo del template o quando viene dedotto il tipo sbagliato.
- Gli argomenti che conducono al fallimento del perfect-forward sono gli inizializzatori a graffe, i puntatori nulli espressi come `0` o `NULL`, i dati in

membro `const static` interi solamente dichiarati, i nomi di funzione template e overloaded e i campi bit.

---

[11] L' Elemento 25 spiega che ai riferimenti universali dovrebbe quasi sempre essere applicata std::forward e, al momento della stampa di questo volume, alcuni membri della comunità C++ hanno iniziato a chiamare i riferimenti universali con il nome di *riferimenti forward*.

[12] Fra gli oggetti locali considerabili per questa ottimizzazione, vi è la maggior parte delle variabili locali (come w all'interno di makeWidget), ma anche gli oggetti temporanei creati nell'ambito di un'istruzione return. I parametri di funzione, invece, no. Alcuni distinguono fra l'applicazione dell'ottimizzazione RVO a oggetti locali con nome e senza nome (ovvero temporanei), limitando il termine RVO agli oggetti senza nome e chiamando invece l'ottimizzazione degli oggetti con nome con il termine NRVO (*NamedReturn Value Optimization*).

[13] Si suppone che i bitfield siano disposti dal lsb (il bit meno significativo) al msb (il bit più significativo). Il C++ non lo garantisce, ma spesso i compilatori forniscono un meccanismo che consente ai programmatori di controllare la disposizione dei bit nel campo.

6

Le espressioni Lambda

Le espressioni lambda, o semplicemente “le lambda” sono state una novità dirompente nel campo della programmazione C++. Questo può essere sorprendente, in quanto esse non apportano alcuna nuova potenza espressiva al linguaggio. Tutto ciò che può fare una lambda, potrebbe essere fatto anche a mano, semplicemente digitando del codice in più. Ma le lambda sono un modo così comodo per creare oggetti funzione che l’impatto che hanno avuto sullo sviluppo quotidiano di software C++ è enorme. Senza le lambda, gli algoritmi `_if` STL (`std::find_if`, `std::remove_if`, `std::count_if` e così via) verrebbero impiegati solo con i predicati più banali, mentre da quando sono disponibili le lambda, fioriscono ovunque gli utilizzi di questi algoritmi in condizioni tutt’altro che banali. Lo stesso vale per gli algoritmi che possono essere personalizzati con delle funzioni di confronto (per esempio `std::sort`, `std::nth_element`, `std::lower_bound` e così via). All’esterno della STL, le lambda consentono di creare rapidamente dei cancellatori custom per `std::unique_ptr` e `std::shared_ptr` (vedi gli Elementi 18 e 19), ed essi rendono altrettanto semplice la specifica dei predicati per le variabili di condizione nell’API per thread (vedi Elemento 39). Ma a parte la Libreria Standard, le lambda facilitano la specifica “al volo” delle funzioni callback, delle funzioni di adattamento dell’interfaccia e delle funzioni contestuali per le chiamate one-off. Le lambda rendono pertanto il C++ un linguaggio di programmazione più “confortevole”.

Il lessico delle lambda può talvolta essere fonte di confusione. Ecco un breve riepilogo.

- *Un'espressione lambda* è semplicemente un'espressione. È una parte del codice sorgente. Nella riga

  ```
  std::find_if(container.begin(), container.end(),
               [](int val) { return 0 < val && val < 10; });
  ```

  la lambda è l'espressione evidenziata.

- Una *closure* (letteralmente, "chiusura") è l'oggetto runtime creato da una lambda. A seconda della modalità di cattura, la closure contiene copie o riferimenti ai dati catturati. Nella chiamata a std::find_if precedente, la closure è l'oggetto che viene passato runtime come terzo argomento a std::find_if.

- Una *classe closure* è una classe dalla quale la closure viene istanziata. Ogni lambda fa sì che i compilatori generino una classe closure univoca. Le istruzioni all'interno di una lambda divengono istruzioni eseguibili nelle funzioni membro della relativa classe closure.

Una lambda viene frequentemente utilizzata per creare una closure che viene usata solo come argomento di una funzione. Questo è il caso della chiamata a `std::find_if` precedente. Tuttavia, le closure possono generalmente essere copiate, pertanto è normalmente possibile avere più closure di un certo "tipo closure" corrispondenti a un'unica lambda. Per esempio, nel codice seguente,

```
{
  int x;                                  // x è una variabile
                                          locale

  …

  auto c1 =                               // c1 è una copia della
   [x](int y) { return x * y > 55; };    // closure prodotta
                                          // dalla lambda

  auto c2 = ci;                           // c2 è una copia di ci
```

```
    auto c3 = c2;                        // c3 è una copia di c2

    …

}
```

`c1`, `c2` e `c3` sono tutte copie della closure prodotta dalla lambda.

Informalmente, è perfettamente accettabile parlare indistintamente di lambda, closure e classi closure. Ma negli Elementi che seguono, spesso è importante distinguere ciò che esiste durante la compilazione (le lambda e le classi closure), ciò che esiste runtime (le closure) e le relazioni che legano questi elementi.

Elemento 31 – Evitare le modalità di cattura di default

In C++11 esistono due modalità di cattura di default: per riferimento e per valore. La cattura di default per riferimento può portare ad avere riferimenti pendenti. La cattura di default per valore induce a pensare di essere immuni a questo problema (e non è così) e illude che le closure siano entro-contenute (e non sempre è così).

Questa è sostanzialmente la "teoria" di questo Elemento. Ma a chi sviluppa codice la teoria non basta e dunque partiamo con i rischi della cattura di default per riferimento.

Una cattura per riferimento fa in modo che la closure contenga un riferimento a una variabile locale o a un parametro che è disponibile nel campo di visibilità (scope) in cui è definita la lambda. Se il ciclo di vita di una closure creata da tale lambda supera il ciclo di vita della variabile locale o del parametro, il riferimento presente nella closure risulterà pendente. Per esempio, supponete di avere un container di funzioni di filtraggio, ognuna delle quali accetta un `int` e restituisce un `bool` che indica se il valore passato soddisfa il filtro:

```
using FilterContainer =                     // vedi Elemento 9 per
    std::vector<std::function<bool(int)>>; // "using" e l'Elemento 2
                                            // per std::function
```

```
FilterContainer filters;                    // funzioni di filtraggio
```

Potremmo aggiungere un filtro per i multipli di 5:

```
filters.emplace_back(                        // vedi Elemento 42 per
   [](int value) { return value % 5 == 0; }  // info su
);                                           // emplace back
```

Tuttavia, potremmo essere nelle condizioni di calcolare il fattore runtime, ovvero potremmo non avere la possibilità di specificare direttamente il valore 5 all'interno della lambda. Pertanto, l'aggiunta del filtro potrebbe avere il seguente aspetto:

```
void addDivisorFilter()
{
   auto calc1 = computeSomeValue1();
   auto calc2 = computeSomeValue2();

   auto divisor = computeDivisor(calc1, calc2);

   filters.emplace_back(                            // rischio!
    [&](int value) { return value % divisor == 0; } // il rif
   );                                               // al divisore
}                                                   // è pendente!
```

Questo codice è davvero una mina vagante, un problema che aspetta solo di manifestarsi. La lambda fa riferimento alla variabile locale `divisor`, ma tale variabile cessa di esistere all'uscita di `addDivisorFilter`. Ciò accade immediatamente dopo l'uscita da `filters.emplace_back`, pertanto la funzione aggiunta a `filters` "muore" ancor prima di nascere. L'uso di un filtro causa un comportamento indefinito, praticamente dal momento in cui viene creato.

Ora, lo stesso problema esisterebbe se la cattura per riferimento di `divisor` fosse esplicita,

```
filters.emplace_back(
   [&divisor](int value)                  // rischio! Il rif a
   { return value % divisor == 0; }       // divisor è
);                                        // ancora pendente!
```

ma con una cattura esplicita, è più facile vedere che il funzionamento della lambda dipende dal ciclo di vita di `divisor`. Inoltre, la presenza del nome, “divisor” ci ricorda di garantire che divisor viva almeno quanto le closure della lambda. Questo è un “appunto mentale” più esplicito al vago “assicurati che non vi sia nulla di pendente” comunicato da un semplice `[&]`.

Se sapete che una closure verrà utilizzata immediatamente (per esempio perché viene passata a un algoritmo STL) e non verrà copiata, non vi è alcun rischio che i riferimenti che contiene sopravvivano alle variabili locali e ai parametri dell’ambiente in cui la lambda è stata creata. In tal caso, potreste supporre, non vi sarà alcun rischio di riferimenti pendenti, pertanto nessun motivo di evitare una modalità di cattura di default per riferimento. Per esempio, la nostra lambda di filtraggio potrebbe essere utilizzata solo come argomento di uno `std::all_of` C++, che dice se tutti gli elementi di un intervallo soddisfano una determinata condizione:

```
template<typename C>
void workWithContainer(const C&
container)
{
   auto calci = computeSomeValuel();     // come sopra
   auto calc2 = computeSomeValue2();     // come sopra

   auto divisor = computeDivisor(calc1,  // come sopra
   calc2);

   using ContElemT = typename            // tipo degli
   C::value_type;
                                         // elementi nel
                                         // container
```

```
    using std::begin;                     // per
    using std::end;                       // genericità;
                                          // vedi Elemento 13
    if (std::all_of(                      // se tutti i valori
       begin(container), end(container),  // del container
       [&](const ContElemT& value)        // sono multipli
       { return value % divisor == 0; })  // del divisore...
     ) {
                                          // allora sono...
   } else {
     …                                    // almeno uno
   }                                      // non lo è...
}
```

È vero, questa soluzione è sicura, ma di una sicurezza per certi versi precaria. Se la lambda dovesse essere utile in altri contesti (per esempio come funzione da aggiungere al container `filters`) e venisse copiata-e-incollata in un contesto in cui la sua closure potrebbe sopravvivere a divisor, ci si troverebbe nuovamente con un riferimento pendente e non vi sarebbe nulla nella clausola di cattura a ricordare di svolgere l'analisi del ciclo di vita su divisor.

Sul lungo termine, elencare esplicitamente le variabili locali e i parametri da cui dipende una lambda è semplicemente una questione di migliore ingegneria del software.

A proposito, in C++14, la possibilità di utilizzare `auto` nelle specifiche del parametro della lambda consente di semplificare il codice. Il typedef `ContElemT` può essere eliminato e la condizione `if` può essere rielaborata nel seguente modo:

```
if (std::all_of(begin(container), end(container),
         [&](const auto& value) // C++14
         { return value % divisor == 0; }))
```

Un modo per risolvere il nostro problema del `divisor` sarebbe quello di impiegare la modalità di cattura di default per valore. In pratica, potremmo aggiungere la lambda a `filters` nel seguente modo:

```
filters.emplace_back(                         // ora
   [=](int value) { return value %            // divisor
   divisor == 0; }
);                                            // non può
                                              // “pendere”
```

La soluzione basterebbe per questo esempio, ma, in generale, la cattura di default per valore non garantisce affatto che non sorgano problemi di riferimenti pendenti. Il problema è che se si cattura un puntatore per valore, si copia il puntatore nelle closure che derivano dalla lambda, ma non si impedisce che il codice all’esterno della lambda possa cancellare il puntatore, facendo sì che le sue copie divengano pendenti.

“Ma questo non può accadere assolutamente!”, protesterete. “Avendo letto il Capitolo 4, so di poter contare sui puntatori smart. Ormai solo i vecchi programmatori C++98 usano i puntatori standard e `delete`.” Questo può essere vero, ma è irrilevante, perché, in realtà, tutti usano i puntatori standard, i quali possono essere cancellati anche a vostra insaputa. Nel codice sorgente, è difficile accorgersi quando ciò può accadere.

Supponete che una delle operazioni svolte dagli oggetti `Widget` sia sommare nuove voci al container di filtri:

```
class Widget {
public:
    …                                         // costruttori ecc
    void addFilter() const;                   // aggiunge una voce ai
                                              filtri

private:
    int divisor;                              // usato nel filtro di
                                              Widget
};
```

Si potrebbe definire `Widget::addFilter` così:

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
```

```
  );
)
```

A prima vista, questo può sembrare codice assolutamente sicuro. La lambda dipende da `divisor`, ma la modalità di cattura di default per valore garantisce che `divisor` venga copiato in ogni closure derivata dalla lambda. Non è così?

No, non è così! Proprio per niente! Assolutamente sbagliato. Drammaticamente sbagliato.

Le catture si applicano solo alle variabili locali non statiche (compresi i parametri) entro il campo di visibilità in cui viene creata la lambda. Nel corpo di `Widget::addFilter`, `divisor` non è una variabile locale, è un dato membro della classe `Widget`. Non può essere catturato. Pertanto, se la modalità capture di default viene eliminata, il codice non verrà compilato:

```
void Widget::addFilter() const
{
   filters.emplace back(                       // errore!
    [](int value) { return value %             // divisor
    divisor == 0; }
  );                                           // non
}                                              // disponibile
```

Inoltre, se viene fatto un tentativo di catturare esplicitamente divisor (per valore o per riferimento, la sostanza non cambia), la cattura non verrà compilata, poiché divisor non è né una variabile locale né un parametro:

```
void Widget::addFilter() const
{
   filters.emplace_back(
    [divisor](int value)                       // errore! Nessun divisor
    { return value % divisor == 0; }           // locale da catturare
  );
}
```

Pertanto, se la cattura di default per valore non cattura `divisor`, ma, senza la clausola di cattura di default per valore, il codice non può essere compilato, come uscirne?

La spiegazione conta sull’utilizzo implicito di un puntatore standard: `this`. Ogni funzione membro non statica ha un proprio puntatore `this`, che è sottinteso ogni volta che si menziona un dato membro della classe. All’interno di ogni funzione membro di `Widget`, per esempio, i compilatori sostituiscono internamente gli utilizzi di `divisor` con `this->divisor`. Nella versione di `Widget::addFilter` con una cattura di default per valore,

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }

);
}
```

ciò che viene catturato è il puntatore `this` di `Widget`, non `divisor`. I compilatori trattano il codice come se fosse scritto nel seguente modo:

```
void Widget::addFilter() const
{
    auto currentObjectPtr = this;

    filters.emplace_back(
    [currentObjectPtr](int value)
    { return value % currentObjectPtr->divisor == 0; }
  );
}
```

Comprendere questo equivale a capire che l’utilizzabilità delle closure che derivano da questa lambda è legata al ciclo di vita dello specifico `Widget` per il quale le closure contengono una copia del puntatore `this`. In particolare, considerate il seguente codice che, in base a quanto detto nel Capitolo 4, utilizza i puntatori solo in versione smart:

```
using FilterContainer =                    // come prima
   std::vector<std::function<bool(int)>>;


FilterContainer filters;                   // come prima


void doSomeWork()
{
   auto pw =                               // create Widget; vedi
      std::make_unique<Widget>();
```

```
                                          // Elemento 21 per
                                          // std::make_unique

    pw->addFilter();                      // aggiunge un filtro che usa
                                          // Widget::divisor
    …
}                                         // distrugge Widget; ora filters
                                          // contiene un puntatore pendente!
```

Quando viene eseguita una chiamata a `doSomeWork`, viene creato un filtro che dipende dall'oggetto `Widget` prodotto da `std::make_unique`, per esempio, un filtro che contiene una copia di un puntatore a tale `Widget` (il puntatore `this` del `Widget`). Questo filtro viene aggiunto a `filters`, ma al termine di `std::make_unique`, il `Widget` viene distrutto dallo `std::unique_ptr` che gestisce il suo ciclo di vita (vedi Elemento 18). Da quel punto in poi, `filters` contiene una voce con un puntatore pendente.

Questo specifico problema può essere risolto eseguendo una copia locale del dato membro che si vuole catturare e poi catturando tale copia:

```
void Widget::addFilter() const
{
   auto divisorCopy = divisor;          // copia un dato membro

   filters.emplace_back(
    [divisorCopy] (int value)           // cattura la copia
    { return value % divisorCopy == 0; }// usa la copia
  );
}
```

Per completezza, seguendo questo approccio, funzionerebbe anche la cattura di default per valore,

```
void Widget::addFilter() const
{
   auto divisorCopy = divisor;           // copia un dato membro

   filters.emplace back(
    [=] (int value)                      // cattura la copia
    { return value % divisorCopy == 0; }// usa la copia
   );
}
```

ma perché tentare la fortuna? La modalità di cattura di default è esattamente ciò che ha fatto sì che catturaste accidentalmente this quando pensavate di catturare divisor.

In C++14, un modo migliore per catturare un dato membro consiste nell'utilizzare la cattura lambda generalizzata (vedi Elemento 32):

```
void Widget::addFilter() const
{
   filters.emplace back(                 // C++14:
    [divisor = divisor](int value)       // copia divisor nella
                                         closure
    { return value % divisor == 0; }     // usa la copia
  );
}
```

Non esiste però una modalità di cattura di default per una cattura lambda generalizzata, pertanto anche in C++14, rimane valido il consiglio di questo Elemento, ovvero quello di evitare le modalità di cattura di default.

Un ulteriore difetto della cattura di default per valore è il fatto che suggerisce che le closure corrispondenti siano entro-contenute e isolate rispetto ai cambiamenti ai dati che si verificano all'esterno delle closure stesse. In generale, questo non è vero, poiché le lambda possono dipendere non solo dalle variabili locali e dai parametri (che possono essere catturati), ma anche da oggetti con *durata di memorizzazione statica*. Tali oggetti sono definiti con campo di visibilità (scope) globale o del namespace o sono dichiarati `static` all'interno di classi, funzioni o file. Questi oggetti possono essere utilizzati all'interno delle

lambda, ma non possono essere catturati. Tuttavia, le specifiche della modalità di cattura di default per valore possono dare l'impressione che essi lo siano. Considerate questa versione riveduta della funzione `DivisorFilter` che abbiamo visto in precedenza:

```
void addDivisorFilter()
{
   static auto calc1 =
   computeSomeValue1();                    // ora static
   static auto calc2 =
   computeSomeValue2();                    // ora static

   static auto divisor =                   // ora static
    computeDivisor(calc1, calc2);

   filters.emplace back(
    [=](int value)                         // non cattura nulla!
    { return value % divisor == 0; }       // fa riferimento alla
                                           static precedente
  );

  ++divisor;                               // modifica divisor
}
```

Un lettore casuale di questo codice potrebbe essere fuorviato nel vedere `[=]` e pensare: "Ok, la lambda crea una copia di tutti gli oggetti che utilizza e pertanto è entro-contenuta". Ma non è così. Questa lambda non utilizza alcuna variabile locale non statica, pertanto non cattura nulla. Al contrario, il codice della lambda fa riferimento alla variabile statica `divisor`. Quando, alla fine di ogni chiamata a `addDivisorFilter`, `divisor` viene incrementata, ogni lambda che sia stata aggiunta a `filters` tramite questa funzione esibirà un nuovo comportamento (corrispondente al nuovo valore di `divisor`). In termini pratici, questa lambda cattura `divisor` per riferimento, una contraddizione diretta a ciò che la clausola di cattura di default per valore sembra implicare. Se vi terrete alla larga dalle clausole di cattura di default per valore, non correrete il rischio che il vostro codice possa essere interpretato erroneamente in questo modo.

Argomenti da ricordare

- La cattura di default per riferimento può generare riferimenti pendenti.
- La cattura di default per valore è suscettibile ai puntatori pendenti (in particolare `this`) e suggerisce, in modo fuorviante, che le lambda siano entro-contenute.

Elemento 32 – Usare la cattura iniziale per spostare gli oggetti nelle closure

Talvolta né la cattura per valore né la cattura per riferimento sono ciò di cui abbiamo realmente bisogno. Se avete un oggetto move-only (per esempio uno `std::unique_ptr` o uno `std::future`) che volete trasformare in una closure, il C++11 non offre alcun modo per farlo. Se avete un oggetto che sarebbe costoso copiare ed economico spostare (come la maggior parte dei container della Libreria Standard), e volete trasformare tale oggetto in una closure, probabilmente preferirete spostarlo piuttosto che copiarlo. Tuttavia, ancora una volta, il C++11 non offre alcun modo per farlo.

Ma stiamo parlando del C++11. Il C++14 è tutta un'altra cosa. Offre un supporto diretto per lo spostamento degli oggetti all'interno delle closure. Se i vostri compilatori sono compatibili C++14, su con il morale e continuate a leggere. Se invece state ancora lavorando con i compilatori C++11, anche voi dovreste tirarvi su di morale e continuare a leggere, perché esistono dei modi per simulare la cattura per spostamento anche in C++11.

L'assenza della cattura per spostamento ha suscitato lamentele fin dal momento in cui il C++11 è stato adottato. Il rimedio immediato sarebbe stato quello di aggiungerla al C++11, ma il Comitato di Standardizzazione ha scelto un percorso differente. Ha introdotto un nuovo meccanismo di cattura che è talmente flessibile, che la cattura per spostamento è solo uno dei trucchi che è in grado di svolgere. La nuova funzionalità si chiama *cattura iniziale* (init capture). Può fare praticamente tutto ciò che possono fare le forme di cattura C++11 e molto altro ancora. L'unica cosa che non potete esprimere con una cattura iniziale è una modalità di cattura di default, ma, come spiega l'Elemento 31, dovreste tenervene, comunque, alla larga. Per le situazioni coperte dalle catture

C++11, la sintassi della cattura iniziale è un po’ prolissa, pertanto nei casi in cui una cattura C++11 svolga il lavoro richiesto, è perfettamente ragionevole utilizzarla.

L’uso della cattura iniziale consente di specificare:

1. **il nome di un dato membro** nella classe closure generata dalla lambda e
2. **un’espressione** che inizializza tale dato membro.

Ecco come potete utilizzare la cattura iniziale per trasformare uno std::unique_ptr in una closure:

```
class Widget {                          // un tipo utile
public:
   …

   bool isValidated() const;
   bool isProcessed() const;
   bool isArchived() const;

private:
   …
};
auto pw = std::make_unique<Widget>();   // crea Widget
                                        // vedi Elemento 21 per
                                        info su
                                        // std::make_unique

   …                                    // configura *pw

                                        // inizializza il dato
auto func = [pw = std::move(pw)]        membro
      { return pw->isValidated()        // nella closure con
              && pw->isArchived(); };   // std::move(pw)
```

Il testo evidenziato indica la cattura iniziale. A sinistra del segno = si trova il nome del dato membro nella classe closure che state specificando; a destra si trova l’espressione di inizializzazione. È interessante notare che il campo di

visibilità a sinistra del segno = è differente dal campo di visibilità a destra del segno. Il campo di visibilità del lato sinistro è quello della classe closure. Il campo di visibilità del lato destro è lo stesso in cui è stata definita la lambda. Nell'esempio precedente, il nome `pw` a sinistra del segno = fa riferimento a un dato membro della classe closure, mentre il nome `pw` sul lato destro fa riferimento all'oggetto dichiarato sopra la lambda, ovvero la variabile inizializzata dalla chiamata a `std::make_unique`. Pertanto, `pw = std::move(pw)` significa "crea nella closure un dato membro `pw` e inizializzalo con il risultato dell'applicazione di `std ::move` alla variabile locale `pw`".

Come di consueto, il codice contenuto nel corpo della lambda si trova nel campo di visibilità della classe closure, pertanto gli utilizzi di `pw`, qui, fanno riferimento al dato membro della classe closure.

Il commento `"configura pw"` nell'esempio precedente indica che dopo che il `Widget` è stato creato da `std::make_unique` e prima che lo `std::unique_ptr` a tale `Widget` venga catturato dalla lambda, il `Widget` viene modificato in qualche modo. Se questa configurazione non è necessaria, ovvero se il `Widget` creato da `std::make_unique` è in uno stato adatto a essere catturato dalla lambda, la variabile locale `pw` non è necessaria, poiché il dato membro della classe closure può essere inizializzato direttamente da std::make_unique:

```
auto func = [pw =                          // inizializza il dato
std::make_unique<Widget>() ]               membro
      { return pw->isValidated()           // nella closure
              && pw->isArchived(); };      // con il risultato della
                                           chiamata
                                           // a make_unique
```

Ciò dovrebbe chiarire che il concetto di "cattura" del C++14 è molto più generalizzato rispetto a quello del C++11, poiché in C++11 non è possibile catturare il risultato di un'espressione. Questo è il motivo per cui la cattura iniziale è chiamata anche *cattura lambda generalizzata.*

Ma cosa accade se uno o più dei compilatori che utilizzate non offrono il supporto per la cattura iniziale C++14? Come ottenere la cattura per spostamento in un linguaggio che non offre il supporto per la cattura per spostamento?

Ricordate che un'espressione lambda non è che un modo per far sì che una classe venga generata e poi venga creato un oggetto di tale tipo. Una lambda non

offre nulla di più di ciò che potrebbe essere fatto anche manualmente. Il codice d'esempio C++14 che abbiamo appena visto, per esempio, potrebbe essere scritto in C++11 nel seguente modo:

```
class IsValAndArch {                        // "viene convalidata
public:                                     // e archiviata"
   using DataType =
   std::unique_ptr<Widget>;

   explicit IsValAndArch(DataType&& ptr)    // vedi Elemento 25
   : pw(std::move(ptr)) {}                  // sull'uso di std::move

   bool operator()() const
   { return pw->isValidated() && pw-
   >isArchived(); }

private:
    DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

Certamente, si tratta di un po' più di lavoro rispetto alla scrittura della la lambda, ma ciò non cambia il fatto che se volete una classe C++11 che supporti l'inizializzazione per spostamento dei propri dati membro, l'unica cosa che vi separa dal risultato è un po' di lavoro in più alla tastiera.

Se però volete rimanere nell'ambito delle lambda (e data la loro comodità, probabilmente è un'ottima scelta), la cattura per spostamento può essere simulata anche in C++11,

1. **spostando l'oggetto da catturare in un oggetto funzione prodotto da std:: bind** e
2. **fornendo alla lambda un riferimento all'oggetto "catturato".**

Se conoscete `std::bind`, il codice è piuttosto semplice. Se invece non la conoscete, il codice richiede un po' di abitudine, ma vale la pena di studiarlo.

Supponete di voler creare uno `std::vector` locale, di dovervi inserire un set

appropriato di valori e di doverlo spostare in una closure. In C++14, la cosa è molto semplice:

```
std::vector<double> data;               // oggetto da spostare
                                        // nella closure

…                                       // inserimento dei dati

auto func = [data = std::move(data)]    // cattura iniziale C++14
            { /* uso dei dati */ };
```

In questo codice ho evidenziato gli elementi chiave: il tipo dell'oggetto che si vuole spostare (`std::vector<double>`), il nome di tale oggetto (`data`) e l'espressione di inizializzazione per la cattura iniziale (`std::move(data)`). L'equivalente C++11 è il seguente, dove ho evidenziato gli stessi elementi:

```
std::vector<double> data;               // come sopra

                                        // come sopra

auto func =
  std::bind(                            // Emulazione C++11
    [](const std::vector<double>& data) // della cattura iniziale
    { /* uso dei dati */ },
    std::move(data)
);
```

Come le espressioni lambda, `std::bind` produce oggetti funzione. Gli oggetti funzione restituiti da `std::bind` possono essere chiamati *oggetti bind*. Il primo argomento di `std::bind` è l'oggetto richiamabile. Gli argomenti successivi rappresentano i valori da passare a tale oggetto.

Un oggetto bind copia tutti gli argomenti passati a `std::bind`. Per ogni argomento lvalue, l'oggetto corrispondente dell'oggetto bind viene costruito per copia. Per ogni oggetto rvalue, viene costruito per spostamento. In questo esempio, il secondo argomento è un rvalue (il risultato di `std::move`, vedi

Elemento 23), pertanto `data` viene costruito per spostamento all'interno dell'oggetto bind. Questa costruzione per spostamento è il problema dell'emulazione della cattura per spostamento, poiché lo spostamento di un rvalue in un oggetto bind è il modo in cui risolviamo l'impossibilità di spostare un rvalue in una closure C++11.

Quando viene "richiamato" l'oggetto bind (ovvero viene richiamato il suo operatore di chiamata a funzione), gli argomenti che conserva vengono passati all'oggetto richiamabile originariamente passato a `std::bind`. In questo esempio, ciò significa che quando viene richiamata `func` (l'oggetto bind), alla lambda che è stata passata a `std::bind` viene passata come argomento la copia costruita per spostamento di `data` all'interno di `func`.

Questa lambda è la stessa della lambda che useremmo in C++14, tranne un parametro, `data`, aggiunto per corrispondere al nostro oggetto catturato per pseudo-spostamento. Questo parametro è un riferimento lvalue alla copia di `data` contenuta nell'oggetto bind (non è un riferimento rvalue, poiché anche se l'espressione utilizzata per inizializzare la copia di `data`, ovvero `std::move(data)`, è un rvalue, la *copia* di `data` è invece un lvalue). Gli utilizzi di `data` all'interno della lambda opereranno pertanto sulla copia costruita per spostamento di `data` all'interno dell'oggetto bind.

Per default, la funzione membro `operator()` all'interno della classe closure generata da una lambda è `const`. Ciò ha l'effetto di rendere `const` tutti i dati membro della closure, all'interno del corpo della lambda. La copia costruita per spostamento di `data` all'interno dell'oggetto bind, invece, non è `const`, pertanto, per impedire che la copia di `data` venga modificata all'interno della lambda, il parametro della lambda viene dichiarato come riferimento-a-**`-const`.** Se la lambda venisse dichiarata `mutable`, l**'`operator()`** nella sua classe closure non verrebbe dichiarato `const` e sarebbe appropriato omettere il modificatore `const` nella dichiarazione del parametro della lambda:

```
auto func =
   std::bind(                                    // Emulazione C++11
    [](std::vector<double>& data)                // della cattura iniziale
    mutable
    { /* uso dei dati */ },                      // per lambda mutable
    std::move(data)
```

```
);
```

Poiché un oggetto bind conserva delle copie di tutti gli argomenti passati a `std::bind`, l'oggetto bind del nostro esempio contiene una copia della closure prodotta dalla lambda che rappresenta il suo primo argomento. Il ciclo di vita della closure coincide pertanto con quello dell'oggetto bind. Questo è importante, poiché significa che fintantoché esiste la closure, esiste anche l'oggetto bind contenente l'oggetto catturato per pseudo-spostamento.

Se è la prima volta che usate `std::bind`, potreste dover consultare il vostro testo di riferimento C++11 preferito per far sì che tutti i dettagli di questa discussione si collochino al posto giusto. Ciononostante, dovrebbero essere chiari i seguenti punti.

- Non è possibile costruire per spostamento un oggetto in una closure C++11, mentre è possibile costruire per spostamento un oggetto in un oggetto bind C++11.
- L'emulazione della cattura per spostamento in C++11 è costituita dalla costruzione per spostamento di un oggetto in un oggetto bind, con successivo passaggio (per riferimento) dell'oggetto costruito per spostamento alla lambda.
- Poiché il ciclo di vita dell'oggetto bind coincide con quello della closure, è possibile trattare gli oggetti nell'oggetto bind come se fossero nella closure.

Come secondo esempio d'uso di std::bind per emulare la cattura per spostamento, ecco il codice C++14 che abbiamo visto in precedenza per creare uno std::unique_ `ptr` in una closure:

```
auto func = [pw =
std::make_unique<Widget>()]              // come prima,
    { return pw->isValidated()           // crea pw
           && pw->isArchived(); };       // nella closure
```

Ed ecco l'emulazione C++11:

```
auto func = std::bind(
            [](const std::unique_ptr<Widget>& pw)
           { return pw->isValidated()
```

```
            && pw->isArchived(); },
    std::make_unique<Widget>()
);
```

Ironia della sorte, mentre sto mostrando come utilizzare `std::bind` per aggirare i limiti delle lambda in C++11, nell'Elemento 34 consiglio di utilizzare le lambda piuttosto che `std::bind`. Tuttavia, tale Elemento spiega che vi sono alcuni casi in C++11 in cui `std::bind` può essere utile, e questo è uno di quelli. In C++14, nuove funzionalità come la cattura iniziale e i parametri `auto` eliminano invece questi casi.

Argomenti da ricordare

- Utilizzate la cattura iniziale C++14 per spostare gli oggetti nelle closure.
- In C++11, potete emulare la cattura iniziale tramite classi realizzate a mano o `std::bind`.

Elemento 33 – Usate decltype sui parametri auto&& per inoltrarli con std::forward

Una delle funzionalità più interessanti del C++14 è costituita dalle *lambda generiche*, lambda che usano `auto` nella specifica dei parametri. L'implementazione di questa funzionalità è semplice: l'`operator()` nella classe closure della lambda è un template. Per esempio, data la seguente lambda,

```
auto f = [](auto x){ return func(normalize(x)); };
```

l'operatore di chiamata a funzione della classe closure ha il seguente aspetto:

```
class SomeCompilerGeneratedClassName {
public:
   template<typename T>                      // vedi Elemento 3 perr
   auto operator()(T x) const                // il tipo restituito da
                                             auto
   { return func(normalize(x)); }

  …                                          // altra funzionalità
```

```
};                                          // della classe closure
```

In questo esempio, l'unica cosa che fa la lambda con il suo parametro `x` è inoltrarlo a `normalize`. Se `normalize` tratta gli lvalue in modo differente rispetto agli rvalue, questa lambda non è scritta in modo corretto, poiché passa a `normalize` sempre un lvalue (il parametro `x`), anche se l'argomento che era stato passato alla lambda era un rvalue.

Il modo corretto per descrivere la lambda è far sì che esegua un perfect-forward di `x` a `normalize`. Per farlo, occorre applicare due modifiche al codice. Innanzitutto, `x` deve diventare un riferimento universale (vedi Elemento 24) e, in secondo luogo, deve essere passato a `normalize` tramite `std::forward` (vedi Elemento 25). Si tratta pertanto di modifiche banali:

```
auto f = [](auto&& x)
          { return func(normalize(std::forward<???>(x))); };
```

Ma fra la teoria e la realizzazione c'è una domanda: quale tipo passare a `std::forward`? Occorre pertanto determinare cosa si deve scrivere fra le parentesi angolari al posto dei `???`

Normalmente, quando si impiega il perfect-forwarding, ci si trova in una funzione template che prende un parametro per il tipo, `T`, pertanto basta scrivere `std::forward<T>`. Ma, nella lambda generica, non è disponibile un parametro per il tipo, `T`. Esiste un `T` nel `operator()` template-izzato all'interno della classe closure generata dalla lambda, ma non è possibile farvi riferimento dalla lambda e pertanto non serve a nulla.

L'Elemento 28 spiega che, se a un parametro riferimento universale viene passato un argomento lvalue, il tipo di tale parametro diviene un riferimento lvalue. Se viene passato un rvalue, il parametro diviene un riferimento rvalue. Ciò significa che nella nostra lambda possiamo determinare se l'argomento passato era un lvalue oppure un rvalue, controllando il tipo del parametro `x`. Un modo per farlo consiste nell'utilizzare `decltype` (vedi Elemento 3). Se è stato passato un lvalue, `decltype(x)` produrrà un tipo che è un riferimento lvalue. Se è stato passato un rvalue, `decltype(x)` produrrà un tipo che è un riferimento rvalue.

L'Elemento 28 spiega anche che, quando si richiama `std::forward`, la convenzione stabilisce che l'argomento per il tipo sia un riferimento lvalue per indicare un lvalue, e un non-riferimento per indicare un rvalue. Nella nostra

lambda, se `x` è associato a un lvalue, `decltype(x)` fornirà un riferimento lvalue. Ciò segue la convenzione. Al contrario, se `x` è associato a un rvalue, `decltype(x)` fornirà un riferimento rvalue invece di un classico non-riferimento.

Ma osservate l'implementazione d'esempio C++14 per `std::forward` tratta dall'Elemento 28:

```
template<typename T>                          // nel namespace
T&& forward(typename                          // std
remove_reference<T>::type& param)
{
    return static_cast<T&&>(param);
}
```

Se il codice client vuole eseguire un perfect-forward di un rvalue di tipo `Widget`, istanzia normalmente `std::forward` con il tipo `Widget` (ovvero un tipo non-riferimento) e il template `std::forward` fornirà questa funzione:

```
Widget&& forward(Widget& param)               // istanziazione di
{                                             // std::forward quando
    return static_cast<Widget&&>(param);      // T è Widget
}
```

Ma considerate ciò che accadrebbe se il codice client volesse eseguire un perfect-forward dello stesso rvalue di tipo `Widget`, ma invece di seguire la convenzione di specificare `T` come un tipo non-riferimento, lo specificasse come un riferimento rvalue. Ovvero, considerate ciò che accadrebbe se `T` fosse specificato come `Widget&&`. Dopo l'istanziazione iniziale di `std::forward` e l'applicazione dei `std::remove_reference_t`, ma prima del collasso dei riferimenti (ancora una volta, vedi Elemento 28), `std::forward` avrebbe il seguente aspetto:

```
Widget&& && forward(Widget& param)            // istanziazione di
{                                             // std::forward quando
    return static_cast<Widget&& &&>           // T è Widget&&
    (param);
}                                             // (prima del collassso
                                              // dei riferimenti)
```

Applicando la regola di collasso dei riferimenti (che dice che un riferimento rvalue a un riferimento rvalue diviene un singolo riferimento rvalue), si ha questa istanziazione:

```
Widget&& forward(Widget& param)          // istanziazione di
{                                        // std::forward quando
    return static_cast<Widget&&>(param); // T è Widget&&
}                                        // (dopo il collasso
                                         // dei riferimenti)
```

Se confrontate questa istanziazione con quella che risulta quando `std::forward` viene richiamata con `T` uguale a `Widget`, vedrete che sono identiche. Ciò significa che l'istanziazione di `std::forward` con un tipo che è un riferimento rvalue fornisce lo stesso risultato di un'istanziazione con un tipo che è un non-riferimento.

Questa è una meravigliosa notizia, poiché `decltype(x)` fornisce un tipo riferimento rvalue quando al parametro `x` della nostra lambda viene passato come argomento un rvalue. Abbiamo detto in precedenza che, quando alla lambda viene passato un rvalue, `decltype(x)` fornisce il tipo consueto da passare a `std::forward` e ora comprendiamo che, per gli rvalue, `decltype(x)` fornisca un tipo da passare a `std::forward` che non è convenzionale, ma ciononostante, fornisce lo stesso risultato di un tipo convenzionale. Pertanto, sia per gli lvalue sia per gli rvalue, il passaggio di `decltype(x)` a `std::forward` fornisce il risultato desiderato. La nostra lambda perfect-forward può pertanto essere scritta nel seguente modo:

```
auto f =
    [](auto&& param)
    {
        return
          func(normalize(std::forward<decltype(param)>(param)));
        };
```

Da questo punto di partenza, basta poco per realizzare una lambda perfect-forward che accetta non solo unico parametro, ma un numero qualsiasi di parametri, poiché in C++14 le lambda possono anche essere variadic:

```
auto f =
    [](auto&&... params)
    {
        return
```

```
        func(normalize(std::forward<decltype(params)>
        (params)...));
    };
```

Argomenti da ricordare

- Utilizzate `decltype` sui parametri `auto&&` per inoltrarle con `std::forward`.

Elemento 34 – Preferite le lambda a std::bind

`std::bind` è l'evoluzione C++11 di `std::bind1st` e `std::bind2nd` del C++98, ma, informalmente, ha fatto parte della Libreria Standard fin dal 2005. Si tratta del momento in cui il Comitato di Standardizzazione ha adottato un documento noto come TR1, che ha introdotto le specifiche di `bind` (in TR1, `bind` era in un namespace differente, pertanto era `std::tr1::bind`, non `std::bind` e alcuni dettagli di interfacciamento erano differenti). Questa sua storia fa sì che alcuni programmatori abbiano un decennio o più di esperienza nell'utilizzo di `std::bind`. Se siete uno di loro, potreste essere riluttanti ad abbandonare uno strumento che vi ha servito così bene. Questo è comprensibile, ma in questo caso, il cambiamento è in positivo, poiché in C++11 le lambda sono quasi sempre una scelta migliore rispetto a `std::bind`. Quanto al C++14, il caso delle lambda non è semplicemente più potente: surclassa ogni altro approccio.

Questo Elemento presuppone che abbiate familiarità con `std::bind`. In caso contrario, dovreste studiarne il funzionamento prima di proseguire la lettura. Tale conoscenza è comunque di grande utilità, poiché non potete mai sapere quando vi capiterà di incontrare degli utilizzi di `std::bind` in una base di codice che vi toccherà di leggere o modificare.

Come abbiamo già visto nell'Elemento 32, farò riferimento agli oggetti funzione restituiti da `std::bind` come *oggetti bind.*

Il motivo più importante per preferire le lambda a `std::bind` è il fatto che le lambda sono più leggibili. Supponete, per esempio, di avere una funzione per configurare un allarme acustico:

```
// typedef per un determinato istante (vedi Elemento 9 per la
sintassi)
using Time = std::chrono::steady_clock::time_point;
```

```
// vedi Elemento 10 per enum class
enum class Sound { Beep, Siren, Whistle };

// typedef per la durata
using Duration = std::chrono::steady_clock::duration;

// al tempo t, emetti il suono s per la durata d
void setAlarm(Time t, Sound s, Duration d);
```

Inoltre, supponete che a un certo punto del programma, abbiamo determinato di voler predisporre un allarme che scocchi un'ora dopo l'avvio e che rimanga acceso per 30 secondi. L'emissione acustica rimane però in sospeso. Possiamo scrivere una lambda che modifica l'interfaccia di `setAlarm` in modo che debba solo essere specificato un suono:

```
// setSoundL ("L" per "lambda") è un oggetto funzione che consente
// di specificare un suono per l'allarme di 30 sec che scatta
// con un'ora di ritardo
auto setSoundL =
  [](Sound s)
{
    // rende disponibili i componenti di std::chrono senza
    qualifica using namespace std::chrono;

    setAlarm(steady clock::now() + hours(1),   // alarme da lanciare
             s,                                // dopo un'ora
             seconds(30));                     // per 30 secondi
};
```

Ho evidenziato la chiamata a `setAlarm` all'interno della lambda. Questa è una chiamata a funzione dall'aspetto normale e anche chi non abbia una grande esperienza nell'uso delle lambda può vedere che il parametro `s` passato alla lambda viene passato come argomento a `setAlarm`.

Possiamo semplificare questo codice in C++14, avvalendoci dei suffissi standard per i secondi (`s`), i millisecondi (`ms`), le ore (`h`) e così via, che si basano sul supporto C++11 dei letterali definiti dall'utente. Questi suffissi sono implementati nel namespace `std ::literals` e, pertanto, il codice precedente

può essere riscritto nel seguente modo:

```
auto setSoundL =
   [](Sound s)
   {
    using namespace std::chrono;
    using namespace std::literals;      // suffissi C++14

    setAlarm(steady clock::now() + 1h,  // con lo stesso
      s,                                // significato
      30s);                             // come sopra
};
```

Il nostro primo tentativo di scrivere la chiamata `std::bind` corrispondente è il seguente. Contiene un errore che correggeremo subito, ma il codice corretto è ben più complicato e poi questa versione semplificata evidenzia alcuni argomenti importanti:

```
using namespace std::chrono;            // come sopra
using namespace std::literals;

using namespace std::placeholders;      // necessario per l'uso di
                                        // " 1"

auto setSoundB =                        // "B" per "bind"
   std::bind(setAlarm,
    steady_clock::now() + 1h,           // errato! Vedi sotto
    _1,
    30s);
```

Mi piacerebbe evidenziare la chiamata a `setAlarm` qui, come ho fatto nella lambda, ma non vi è alcuna chiamata da evidenziare. Chi legge questo codice deve semplicemente sapere che richiamando `setSoundB` si richiama `setAlarm` con il tempo e la durata specificati nella chiamata a `std::bind`. Per i più inesperti, il segnaposto `_1` è sostanzialmente una "magia", ma anche i lettori più esperti devono farsi una mappa mentale che va dal numero di questo segnaposto

alla sua posizione nell'elenco dei parametri di `std::bind` per comprendere che il primo argomento in una chiamata `setSoundB` viene passato come secondo argomento `setAlarm`. Il tipo di questo argomento non è identico nella chiamata `std::bind`, pertanto i lettori devono consultare la dichiarazione di `setAlarm` per determinare quale tipo di argomento passare a `setSoundB`.

Ma, come ho detto, il codice non è affatto corretto. Nella lambda, è chiaro che l'espressione `steady_clock::now() + 1h` è un argomento per `setAlarm`. Verrà valutato quando verrà richiamata `setAlarm`. Ciò ha perfettamente senso: vogliamo che l'allarme venga lanciato un'ora dopo la chiamata di `setAlarm`. Nella chiamata a `std::bind`, invece, `steady_clock::now() + 1h` viene passato come argomento a `std::bind`, non a `setAlarm`. Ciò significa che l'espressione verrà valutata quando verrà richiamata `std::bind` e l'ora risultante da tale espressione verrà memorizzata all'interno dell'oggetto bind risultante. Come conseguenza, l'allarme verrà impostato per essere lanciato un'ora *dopo la chiamata a `std::bind`*, non un'ora dopo la chiamata a `setAlarm`!

La correzione del problema richiede che venga detto a `std::bind` di ritardare la valutazione dell'espressione finché non verrà richiamata `setAlarm` e il modo per farlo è nidificare una seconda chiamata a `setAlarm` all'interno della prima:

```
auto setSoundB =
  std::bind(setAlarm,
    std::bind(std::plus<>(), steady_clock::now(), 1h),
    _1,
    30s);
```

Se conoscete il template `std::plus` del C++98, potreste essere sorpresi di vedere che in questo codice non venga specificato alcun tipo fra le parentesi angolari, ovvero il codice contiene `std::plus<>` e non `std::plus<`*tipo*`>`. In C++14, l'argomento di tipo template per i template dell'operatore standard può essere omesso, pertanto non vi è alcuna necessità di fornirlo qui. Il C++11 non offre questa funzionalità e, pertanto, in C++11, il `std::bind` equivalente alla lambda è:

```
using namespace std::chrono;                    // come sopra
using namespace std::placeholders;


auto setSoundB =
   std::bind(setAlarm,
```

```
    std::bind(std::plus<steady_clock::time_point>
    (),
         steady_clock::now(),
         hours(1)),
    _1,
    seconds(30));
```

A questo punto, se la lambda non vi sembra molto più interessante, probabilmente dovete mettervi gli occhiali.

Quando `setAlarm` subisce overloading, sorge un nuovo problema. Supponete che vi sia un overload che accetta un quarto parametro che specifica il volume dell’allarme:

```
enum class Volume { Normal, Loud, LoudPlusPlus };
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

La lambda continuerà a funzionare come prima, poiché la risoluzione dell’overload sceglie la versione a tre argomenti di setAlarm:

```
auto setSoundL =                          // come prima
   [](Sound s)
   {
    using namespace std::chrono;

    setAlarm(steady_clock::now() + 1h,    // OK, richiama la
          s,                              // versione a tre argomenti
          30s);                           // di setAlarm
   };
```

Ma `std::bind`, a questo punto, non verrà più compilata:

```
auto setSoundB =                          // errore! Quale
   std::bind(setAlarm,                    // setAlarm?
    std::bind(std::plus<>(),
            steady clock::now(),
           1h),
```

```
    _1,
    30s);
```

Il problema è che i compilatori non hanno alcun modo per determinare quale delle due funzioni `setAlarm` dovrebbero passare a `std::bind`. Tutto ciò che hanno è un nome di funzione, il quale, da solo, è ambiguo.

Per rendere compilabile la chiamata a `std::bind`, `setAlarm` deve essere convertita nel tipo puntatore a funzione corretto:

```
using SetAlarm3ParamType = void(*)(Time t,
Sound s, Duration d);

auto setSoundB =                                    // ora
   std::bind( static_cast<SetAlarm3ParamType>(      // funziona
   setAlarm),
      std::bind(std::plus<>(),
            steady_clock::now(),
            1h),
   _1,
   30s);
```

Ma ciò fa sorgere un'altra differenza fra le lambda e `std::bind`. All'interno dell'operatore di chiamata a funzione per `setSoundL` (ovvero l'operatore di chiamata a funzione della classe closure della lambda), la chiamata a `setAlarm` è una normale chiamata a funzione, che può essere inserita inline dai compilatori nel modo consueto:

```
setSoundL(Sound::Siren);          // il corpo di setAlarm
                                  // può essere inserito inline qui
```

La chiamata a `std::bind`, invece, passa un puntatore a funzione a `setAlarm` e ciò significa che, all'interno dell'operatore di chiamata a funzione per `setSoundB` (ovvero l'operatore di chiamata a funzione dell'oggetto bind), la chiamata `setAlarm` ha luogo attraverso un puntatore a funzione. I compilatori eseguono con meno probabilità l'inserimento inline delle chiamate a funzione tramite

puntatore a funzione e ciò significa che le chiamate a `setAlarm` attraverso `setSoundB` avranno meno probabilità di essere inserite inline rispetto a quelle attraverso setSoundL:

```
setSoundB(Sound::Siren);                    // il corpo di setAlarm ha meno
                                            // probabilità di essere messo inline qui
```

È pertanto possibile che l'utilizzo di lambda generi codice più veloce rispetto all'utilizzo di `std::bind`.

L'esempio di `setAlarm` coinvolge un'unica, semplice, chiamata a funzione. Se avete bisogno di fare qualcosa di più complesso, le cose si complicano ulteriormente, a favore delle lambda. Per esempio, considerate la seguente lambda C++14, che restituisce il fatto che il proprio argomento sia compreso fra un valore minimo (`lowVal`) e un valore massimo (`highVal`), dove `lowVal` e `highVal` sono variabili locali:

```
auto betweenL =
   [lowVal, highVal]
   (const auto& val)                        // C++14
   { return lowVal <= val && val <= highVal; };
```

`std::bind` può esprimere la stessa cosa, ma il costrutto è un classico esempio di programmazione oscurantista:

```
using namespace std::placeholders;             // come sopra

auto betweenB =
   std::bind(std::logical_and<>(),             // C++14
        std::bind(std::less_equal<>(), lowVal, _1),
        std::bind(std::less_equal<>(), _1, highVal));
```

In C++11 dovremmo specificare i tipi che vogliamo confrontare e la chiamata a `std::bind` avrebbe il seguente aspetto:

```
auto betweenB =                                // versione C++11
   std::bind(std::logical_and<bool>(),
        std::bind(std::less_equal<int>(),
    lowVal, _1),
        std::bind(std::less_equal<int>(), _1,
    highVal));
```

Naturalmente, in C++11, la lambda non può prendere un parametro **auto,** pertanto bisogna specificare un tipo:

```
auto betweenL =                         // versione C++11
   [lowVal, highVal]
   (int val)
   { return lowVal <= val && val <= highVal; };
```

A questo punto, spero che converrete che la versione lambda non solo è più compatta, ma è anche più comprensibile e di facile manutenzione.

In precedenza ho rilevato che per coloro che hanno poca esperienza nell'uso di `std::bind`, i segnaposto (`_1`, `_2` e così via) sembrano sostanzialmente "magici". Ma non è solamente il comportamento dei segnaposto a complicare le cose. Supponete di avere una funzione per creare delle copie compresse di `Widget`,

```
enum class CompLevel { Low, Normal, High  // livello di
};
                                          // compressione


Widget compress(const Widget& w,          // crea una copia
           CompLevel lev);                // compressa di w
```

e di voler creare un oggetto funzione che ci consenta di specificare quanto debba essere compresso un oggetto `Widget w`. Questo uso di `std::bind` creerà tale oggetto:

```
Widget w;

using namespace std::placeholders;
```

```
auto compressRateB = std::bind(compress, w, _1);
```

Ora, quando passiamo `w` a `std::bind`, `w` deve essere memorizzata per le successive chiamate a `compress`. È conservata all'interno dell'oggetto `compressRateB`, ma in quale modo? Per valore o per riferimento? Fa una bella differenza, poiché se `w` viene modificata fra la chiamata a `std::bind` e una chiamata a `compressRateB`, la memorizzazione di `w` per riferimento rifletterà la modifica, mentre la memorizzazione per valore no.

La risposta è che viene memorizzata per valore,[14] ma l'unico modo per saperlo consiste nel capire come funziona `std::bind`; non vi è alcun segno di ciò nella chiamata a `std ::bind`. Al contrario, nell'approccio lambda, il fatto che `w` venga catturata per valore o per riferimento è esplicito:

```
auto compressRateL =                    // w viene catturata
   [w](CompLevel lev)                   // per valore; lev viene
   { return compress(w, lev); };        // passata per valore
```

Altrettanto esplicito è il modo in cui i parametri vengono passati alla lambda. Qui è chiaro che il parametro `lev` viene passato per valore. Pertanto:

```
compressRateL(CompLevel::High);         // l'argomento
                                        // viene passato per
                                        valore
```

Ma nella chiamata all'oggetto risultante da `std::bind`, come viene passato l'argomento?

```
compressRateB(CompLevel::High);         // come viene passato
                                        // l'argomento?
```

Ancora una volta, l'unico modo per saperlo consiste nel conoscere il funzionamento di `std::bind`. (La risposta è che tutti gli argomenti passati agli oggetti bind vengono passati per riferimento, poiché l'operatore di chiamata a funzione di tali oggetti usa il perfect-forwarding).

Rispetto alle lambda, pertanto, il codice che utilizza `std::bind` è meno leggibile, meno espressivo e, in generale, meno efficiente. In C++14 non esistono casi d'uso ragionevoli per `std::bind`. In C++11, invece, l'uso di `std::bind` può

essere giustificato in due situazioni ben precise.

- **Cattura per spostamento.** In C++11, le lambda non offrono la cattura per spostamento, ma questa può essere simulata tramite una combinazione di lambda e `std::bind`. Per i dettagli, consultate l'Elemento 32, che spiega anche che in C++14 il supporto delle lambda per la cattura iniziale elimina la necessità di ricorrere all'emulazione.
- **Oggetti funzione polimorfici.** Poiché l'operatore di chiamata a funzione su un oggetto bind utilizza il perfect-forwarding, può accettare argomenti di qualsiasi tipo (a parte le restrizioni sul perfect-forwarding descritte nell'Elemento 30). Ciò può essere utile quando si vuole associare un oggetto a un operatore di chiamata a funzione template-izzato. Per esempio, data questa classe,

```
class PolyWidget {
public:
    template<typename T>
    void operator()(const T& param);
    …
};
```

`std::bind` può associarsi a `PolyWidget` nel seguente modo:

```
PolyWidget pw;
auto boundPW = std::bind(pw,
```

`boundPW può quindi essere richiamata con tipi di argomenti differenti:`

```
boundPW(1930);              // passa l'int a
                            // PolyWidget::operator()

boundPW(nullptr);           // passa nullptr a
                            // PolyWidget::operator()

boundPW("Rosebud");         // passa la stringa
                            letterale a
                            // PolyWidget::operator()
```

Non vi è alcun modo per farlo con una lambda C++11. In C++14, invece, questo viene ottenuto con facilità tramite una lambda con un parametro auto:

```
auto boundPW = [pw](const auto& param)   // C++14
      { pw(param); };
```

Si tratta, naturalmente, di casi limite e destinati a sparire, poiché sono sempre più comuni i compilatori che supportano le lambda C++14.

Quando bind è stato aggiunto in modo non ufficiale al C++ nel 2005, si trattava di un importante miglioramento rispetto ai suoi predecessori "1998". L'aggiunta del supporto lambda al C++11 ha reso `std::bind` del tutto obsoleta, tanto che ai tempi del C++14 non esiste alcuna vera ragione per utilizzarla.

Argomenti da ricordare

- Le lambda sono più leggibili, più espressive e possono essere più efficienti rispetto a `std::bind`.
- Solo in C++11, `std::bind` può essere utile per implementare la cattura per spostamento o per associare gli oggetti agli operatori di chiamata a funzione template-izzati.

---

[14] `std::bind` copia sempre i propri argomenti, ma i chiamanti possono ottenere l'effetto di avere un argomento memorizzato per riferimento, applicandogli std::ref. Il risultato di

```
auto compressRateB = std::bind(compress, std::ref(w), _1);
```

è che compressRateB si comporta come se contenesse un riferimento a w, invece di una copia.

7

L’API per la concorrenza

Uno dei grandi trionfi del C++11 è stato l’inclusione della concorrenza nel linguaggio e nella libreria. I programmatori che conoscono l’uso di altre API dedicate ai thread (per esempio pthreads o i thread di Windows) rimangono talvolta sorpresi dalla relativa “rozzezza” delle funzionalità offerte dal C++, ma questo è dovuto al fatto che gran parte del supporto per la concorrenza offerto dal C++ si trova celato sotto forma di vincoli imposti agli autori di compilatori. Ne derivano alcune garanzie di base offerte dal linguaggio, che, per la prima volta nella storia del C++, consentono ai programmatori di scrivere programmi multi-thread che hanno un comportamento standard su tutte le piattaforme. Questo offre una solida base sulla quale possono essere realizzate librerie molto espressive; gli elementi di concorrenza della Libreria Standard (i task, i future, i thread, i mutex, le variabili di condizione, gli oggetti atomici e molto altro ancora) sono semplicemente l’inizio di ciò che diverrà sicuramente un ricco set di strumenti per lo sviluppo di software C++ concorrente.

Negli Elementi che seguono, tenete sempre in considerazione che la Libreria Standard offre due template per i future: `std::future` e `std::shared_future`. In molti casi, la distinzione non è così importante, pertanto spesso parlerò semplicemente di *future*, nel qual caso, intendo entrambi i tipi.

Elemento 35 – Preferire la programmazione basata su task piuttosto che su thread

Se volete eseguire una funzione `doAsyncWork` in modo asincrono, avete sostanzialmente due possibilità. Potete creare uno `std::thread` e, da lì, eseguire `doAsyncWork`, impiegando pertanto un approccio a thread:

```
int doAsyncWork();

std::thread t(doAsyncWork);
```

Oppure potete passare `doAsyncWork` a `std::async`, adottando una strategia *basata su task*:

```
auto fut =
std::async(doAsyncWork);            // "fut" per "future"
```

In tali chiamate, l'oggetto funzione passato a `std::async` (per esempio `doAsyncWork`) è considerato un task.

L'approccio a task è tipicamente superiore alla sua versione a thread e i piccoli frammenti di codice che abbiamo appena visto dimostrano alcune delle ragioni di questa superiorità. Qui, `doAsyncWork` produce un valore, che, come possiamo ragionevolmente presumere, è ciò cui è interessato il codice che richiama `doAsyncWork`. Con la chiamata a thread non vi è alcun modo lineare per accedervi. Con l'approccio a task, questo è facile, poiché il future restituito da `std::async` offre la funzione `get`. La funzione `get` è ancora più importante se `doAsyncWork` emette un'eccezione, poiché `get` fornisce un accesso anche a questo elemento. Con l'approccio basato su thread, se `doAsyncWork` lancia un'eccezione, il programma si chiude (tramite una chiamata a `std::terminate`).

Una differenza più sostanziale fra la programmazione a thread e a task è il livello di astrazione più elevato offerto dall'approccio a task. Ci libera da tutti i dettagli della gestione del thread, un'osservazione che mi ricorda che devo innanzitutto schematizzare i tre significati della parola "thread" nel software C++ concorrente.

- *I thread hardware* sono thread fisici che svolgono effettivamente il

calcolo. Le architetture delle macchine attuali offrono uno o più thread hardware per ogni core della CPU.

- *I thread software* (chiamati anche thread *del sistema operativo* o, semplicemente, *del sistema*) sono i thread che il sistema operativo[15] gestisce fra tutti i processi e assegna ai thread hardware. Tipicamente è possibile creare più thread software rispetto ai thread hardware, poiché quando un thread software è bloccato (per esempio per un'attività di I/O o in attesa di un mutex o di una variabile di condizione), la produttività del sistema può essere migliorata eseguendo altri thread, che non sono bloccati.
- *Gli `std::thread`* sono degli oggetti all'interno di un processo C++, i quali poggiano sui thread software sottostanti. Alcuni oggetti `std::thread` sono "nulli", ovvero non poggiano su alcun thread software, poiché si trovano in uno stato di costruzione di default (pertanto non hanno alcuna funzione da eseguire), o sono stati spostati (e quindi sarà lo `std::thread` che è la destinazione dello spostamento a poggiarsi sul thread software sottostante), o hanno subito una `join` (la funzione che dovevano eseguire è terminata) o hanno subito un `detach` (la connessione che avevano con il thread software sottostante è stata recisa).

I thread software sono una risorsa limitata. Se tentate di creare più thread di quanti il sistema sia in grado di offrire, viene lanciata un'eccezione `std::system_error`. Questo capita anche se la funzione che volete eseguire non può lanciarne. Per esempio, anche se `doAsyncWork` è noexcept,

```
int doAsyncWork() noexcept;        // vedi Elemento 14 per noexcept
```

questa istruzione può produrre un'eccezione:

```
std::thread t(doAsyncWork);        // lanciata se non sono
                                   // disponibili altri thread
```

Il codice deve in qualche modo gestire questa possibilità. Ma come? Un approccio consiste nell'eseguire `doAsyncWork` sul thread corrente, ma ciò può portare a uno sbilanciamento del carico e, se il thread corrente è relativo all'interfaccia utente, avrà problemi di reattività. Un'altra possibilità consiste

nell’attendere il completamento di un altro thread software, per poi tentare di creare un nuovo `std::thread`, ma è possibile che tali thread siano in attesa di un’azione che `doAsyncWork` dovrebbe svolgere (per esempio, produrre un risultato o notificare una variabile di condizione).

Anche se non esaurite i thread, potreste avere dei problemi di *oversubscription.* Si verifica quando vi sono più thread software pronti a partire (ovvero non bloccati) rispetto ai thread hardware. Quando si verifica questa situazione, lo scheduler di thread (in genere un componente del sistema operativo) gestisce i thread software sui thread hardware operando in time-slice. Quando termina il time-slice di un thread e inizia quello del successivo, viene eseguita una commutazione di contesto. Tale commutazione di contesto incrementa il sovraccarico del sistema di gestione dei thread, la qual cosa può essere particolarmente onerosa nei casi in cui il thread hardware sul quale è programmata l’esecuzione di un determinato thread software si trova in un core differente rispetto a quello in cui si trovava il thread software durante il time-slice precedente. In tal caso, (1) le cache della CPU sono tipicamente vuote di dati di questo thread software (ovvero contengono pochi dati e poche istruzioni utili) e (2) l’esecuzione del “nuovo” thread software finisce per “inquinare” la cache della CPU per i “vecchi” thread che erano in esecuzione su tale core e che, con ogni probabilità, verranno poi eseguiti sempre su questo core.

Evitare la condizione di oversubscription è difficile, poiché il rapporto ottimale fra thread software e hardware dipende dalla frequenza con cui i thread software sono eseguibili e questa situazione può cambiare dinamicamente, per esempio, quando un programma passa da una situazione di intensa attività di I/O a una situazione di intensa attività computazionale. Il miglior rapporto fra thread software e hardware dipende anche dal costo delle commutazioni contestuali e dall’efficacia con cui i thread software usano le cache della CPU. Inoltre, il numero di thread hardware e i dettagli delle cache della CPU (ovvero le loro dimensioni e la loro velocità relativa) dipendono dall’architettura della macchina e pertanto, anche se ottimizziamo l’applicazione per evitare il problema di oversubscription (mantenendo però sempre in attività l’hardware) su una piattaforma, questo non garantisce che la soluzione trovata funzioni anche con altri tipi di macchine.

Le cose si semplificano se scaricate tutti questi problemi su “qualcos’altro”. Questo è proprio lo scopo di `std::async`:

```
auto fut =
std::async(doAsyncWork);            // la gestione dei thread
                                    // ricade sull'implementatore
                                    // della Libreria Standard
```

Questa chiamata delega la responsabilità di gestione dei thread all'implementatore della Libreria Standard del C++. Per esempio, la probabilità di ricevere un'eccezione di esaurimento dei thread si riduce significativamente, poiché questa chiamata probabilmente non ne produrrà mai una. "Come può essere?", potreste chiedervi. "Se richiedo più thread software rispetto a quanti ne può offrire il sistema, perché fa una così grande differenza se li creo con `std::thread` o attraverso `std::async?".` E invece fa una grande differenza, poiché `std::async,` quando viene richiamato in questa forma (ovvero con la politica di lancio di default, vedi Elemento 36), non garantisce che venga creato un nuovo thread software. Piuttosto, permette allo scheduler di predisporre le cose perché la funzione specificata (in questo esempio, `doAsyncWork)` venga eseguita proprio sul thread che richiede i risultati di `doAsyncWork` (ovvero il thread che richiama `get` o `wait` su `fut`) e, normalmente, gli scheduler sfruttano questa libertà d'azione qualora il sistema sia in condizioni di `oversubscription` o abbia esaurito i thread.

Se avete provato a risolvere da soli questa condizione di "eseguirla sul thread che richiede i risultati", probabilmente vi sarete scontrati con problemi di bilanciamento del carico, e questi problemi non spariscono per il solo fatto che si usa `std::async` e che a risolverli debba essere lo scheduler runtime e non voi. Tuttavia, sulle faccende che riguardano il bilanciamento del carico, è probabile che lo scheduler runtime abbia una visione più ampia di ciò che sta accadendo nella macchina rispetto al programmatore, poiché gestisce i thread di tutti processi, non solo del codice che gli abbiamo fatto eseguire.

Con `std::async,` la reattività di un thread dell'interfaccia grafica può essere problematica, poiché lo scheduler non ha alcun modo di sapere quale dei thread ha requisiti di reattività più elevati. In tal caso, potreste voler passare a `std::async` la politica di lancio `std::launch::async.` Ciò garantisce che la funzione che volete eseguire venga effettivamente eseguita su un thread differente (vedi Elemento 36).

Per evitare la condizione di oversubscription, gli scheduler di thread più avanzati impiegano dei thread pool a livello dell'intero sistema e, inoltre, migliorano il

bilanciamento del carico sui core hardware attraverso algoritmi *work-stealing*, di sottrazione del carico. Il C++ standard non richiede l'uso di thread pool e del work-stealing e, per essere onesti, vi sono alcuni aspetti delle specifiche sulla concorrenza del C++11 che ne complicano l'utilizzo più del dovuto. Ciononostante, alcuni produttori sfruttano questa tecnologia nella loro implementazione della Libreria Standard ed è ragionevole aspettarsi che in questo campo i progressi siano continui. Se adottate un approccio a task per la programmazione concorrente, godrete automaticamente dei benefici di questa tecnologia a mano a mano che essa si diffonderà. Se, al contrario, programmate direttamente con `std::thread`, continuerete ad assumervi il carico di gestire sempre l'esaurimento dei thread, il problema della oversubscription e il bilanciamento del carico, per non parlare del modo in cui le vostre specifiche soluzioni a questi problemi andranno a impattare sulle soluzioni implementate dai programmi che vengono eseguiti in altri processi operanti sulla stessa macchina.

Rispetto alla programmazione a thread, l'approccio a task vi solleva dai problemi della gestione manuale dei thread e vi fornisce un modo naturale per esaminare i risultati delle funzioni eseguite in modo asincrono (per esempio, i valori restituiti o le eccezioni). Ciononostante, vi sono alcune situazioni in cui può essere appropriato utilizzare direttamente i thread. Eccone alcune.

- **Dovete accedere all'API dell'implementazione sottostante della gestione dei thread.** L'API C++ per la concorrenza è normalmente implementata utilizzando API specifiche della piattaforma, di basso livello (normalmente pthreads o i thread di Windows). Queste API sono attualmente più ricche rispetto a quelle offerte dal C++. Per esempio, il C++ non ha alcun concetto di priorità o affinità fra thread. Per fornire l'accesso all'API dell'implementazione sottostante per la gestione dei thread, gli oggetti `std::thread` offrono in genere la funzione membro `native_handle`. Non esiste alcuna controparte a questa funzionalità per `std::future` (ovvero per ciò che viene restituito da `std::async)`.
- **Dovete e potete ottimizzare l'utilizzo dei thread per la vostra applicazione.** Questo potrebbe essere il caso, per esempio, in cui stiate sviluppando il software per un server con uno specifico profilo d'esecuzione, il quale verrà impiegato come unico processo significativo su una macchina dotata di caratteristiche hardware fisse.

- **Dovete implementare una tecnologia di gestione dei thread che supera i comportamenti automatici dell'API per la concorrenza del C++**, per esempio una gestione a thread pool su piattaforme in cui l'implementazione del C++ non la offre.

Si tratta, tuttavia, di casi poco comuni. Nella maggior parte dei casi vi troverete a scegliere la gestione a task invece di programmare direttamente con i thread.

Argomenti da ricordare

- L'API `std::thread` non offre alcun modo diretto per ottenere valori restituiti dalle funzioni in esecuzione asincrona e, se queste funzioni lanciano eccezioni, il programma termina.
- La programmazione a thread richiede la gestione manuale dei problemi di esaurimento dei thread, di oversubscription, di bilanciamento del carico e di adattamento alle nuove piattaforme.
- La programmazione a task tramite `std::async` con la politica di lancio di default gestisce la maggior parte di questi problemi in modo automatico.

Elemento 36 – Specificare std::launch::async quando è essenziale l'asincronicità

Quando richiamate `std::async` per eseguire una funzione (o un qualsiasi altro oggetto eseguibile), generalmente intendete eseguire tale funzione in modo asincrono. Ma questo non è necessariamente ciò che state chiedendo di fare a `std::async.` In realtà, state chiedendo alla funzione di operare sulla base della *politica di lancio* di `std::async.` Vi sono due politiche standard, ognuna delle quali è rappresentata da un enumeratore nella enum di `std::async` (vedi Elemento 10 per informazioni sulle `enum).` Supponiamo che a `std::async` venga passata per l'esecuzione una funzione `f`.

- **La politica di lancio `std::launch::async` significa** che `f` deve essere eseguita in modo asincrono, ovvero su un thread differente.
- **La politica di lancio `std::launch::deferred` significa** che `f` può essere

eseguita solo quando viene richiamato un `get` o un `wait` sul future che viene restituito da `std::async`.[16] In pratica, l'esecuzione di `f` viene differita finché non è stata eseguita tale chiamata. Quando viene richiamato `get` o `wait`, `f` verrà eseguita in modo sincrono, ovvero il chiamante si attenderà il termine dell'esecuzione di `f`. Se non vengono richiamate né `get` né `wait`, `f` non verrà mai eseguita.

Sorprendentemente, la politica di lancio di default di `std::async` (quella utilizzata qualora non se ne specifichi esplicitamente una), non è nessuna di queste due. È come se a queste due politiche venisse applicata una sorta di "or". Le due chiamate seguenti hanno esattamente lo stesso significato:

```
auto fut1 = std::async(f);           // esegue f usando
                                     // la politica di lancio
                                     // di default

auto fut2 =
std::async(std::launch::async |      // esegue f in modo
      std::launch::deferred,         // asincrono o
      f);                            // differito
```

La politica di default, pertanto, permette che `f` venga eseguita in modo asincrono oppure sincrono. Come evidenzia l'Elemento 35, questa flessibilità scarica su `std::async` e sui componenti di gestione dei thread della Libreria Standard la responsabilità di creare e distruggere i thread, per evitare il problema della oversubscription e per scopi di bilanciamento del carico. Queste sono attività che rendono davvero comoda la programmazione concorrente con `std::async`.

Ma l'utilizzo degli `std::async` con la politica di lancio di default ha alcune implicazioni interessanti. Dato un thread `t` che esegue la seguente istruzione,

```
auto fut = std::async(f);            // esegue f usando politica di
                                     lancio di default
```

- **Non è possibile prevedere se `f` opererà in modo concorrente con `t`,** poiché `f` potrebbe essere schedulata per essere eseguita in modo differito.
- **Non è possibile prevedere se `f` opererà su un thread differente da**

quello che richiama `get` o `wait` su `fut`. Se tale thread è `t`, l'implicazione è che non è possibile prevedere se `f` verrà eseguita su un thread diverso da `t`.

- **Potrebbe non essere possibile prevedere se `f` verrà, addirittura, eseguita,** poiché potrebbe non essere possibile garantire che `get` o `wait` verranno chiamate su `fut` lungo ogni percorso previsto dal programma.

La flessibilità di scheduling della politica di lancio di default stessa non va troppo d'accordo con l'utilizzo delle variabili `thread_local`, poiché significa che se `f` legge o scrive tale TLS (*Thread-Local Storage*), non è possibile prevedere a quali variabili del thread verrà fatto accesso:

```
auto fut = std::async(f);          // TLS per f forse per
                                   // il thread indipendente, ma
                                   // forse per il thread
                                   // che richiama get o wait su fut
```

Ciò influenza anche i cicli basati su `wait` che utilizzano timeout, poiché la chiamata `wait_for` o `wait_until` su un task (vedi Elemento 35) che è differito fornisce il valore `std::launch::deferred`. Questo significa che il seguente ciclo, che sembra dover terminare, in realtà potrebbe funzionare per sempre:

```
using namespace std::literals;      // per i suffissi di durata
                                    C++14
                                    // vedi Elemento 34

void f()                            // f si ferma per 1 secondo,
{                                   // poi esce
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);           // esegue f in modo asincrono
                                    // (teoricamente)

while (fut.wait_for(100ms) !=       // ciclo finché f ha
    std::future_status::ready)      // terminato l'esecuzione...
```

```
{                                      // che potrebbe non terminare
                                       mai!
  …
}
```

Se `f` opera in modo concorrente con il thread che richiama `std::async` (ovvero se la politica di lancio scelta per `f` è `std::launch::async`), non vi è alcun problema (supponendo che `f`, in qualche modo, termini la propria esecuzione), ma se `f` è differita, `fut.wait_for` restituirà sempre `std::future_status::deferred`. Questo valore non sarà mai uguale a `std::future_status::ready` e dunque il ciclo potrebbe non terminare mai.

Questo genere di bug è facile da sottovalutare durante lo sviluppo e i test dell'unità, poiché potrebbe manifestarsi solo con pesanti condizioni di carico. Vi sono condizioni che spingono la macchina verso problemi di oversubscription o di esaurimento dei thread e proprio queste sono le situazioni in cui è più probabile che avvenga il differimento di un task. Dopotutto, se l'hardware non è in condizioni di oversubscription o di esaurimento dei thread, non vi è alcun motivo per cui il sistema runtime debba eseguire lo scheduling del task per l'esecuzione concorrente.

La correzione è semplice: basta controllare il future corrispondente alla chiamata `std ::async`, per vedere se il task è stato deferito e, in questo caso, evitare di entrare nel ciclo basato su timeout. Sfortunatamente, non esiste un modo diretto per chiedere a un future se il suo task è deferito. Invece, occorre chiamare una funzione basata su timeout, una funzione come `wait_for`. In questo caso, in realtà non vogliamo attendere nulla; vogliamo solo vedere se il valore restituito è `std::future_status::deferred`, pertanto tenete a bada la vostra incredulità e richiamate `wait_for` con un timeout pari a zero:

```
auto fut = std::async(f);          // come sopra

if (fut.wait_for(0s) ==            // se il task è
    std::future_status::deferred)  // differito...
{
                                   // ...usare wait o get su fut
…                                  // per richiamare f in modo
                                   sincrono
```

```
} else {                           // il task non è differito
  while (fut.wait_for(100ms) !=    // il ciclo infinito
      std::future_status::ready)   // non si verifica
      {
                                   // (supponendo che f termini)

                                   // il task non è né differito né
  …                                pronto, quindi
                                   // esegui il lavoro concorrente
                                   finché è pronto
  }
  …                                // fut è pronto
}
```

Il risultato di tutte queste considerazioni è che non si hanno problemi utilizzando `std::async` con la politica di lancio di default per un task, sempre che siano esaudite le seguenti condizioni.

- Il task non deve operare in modo concorrente con il thread che richiama `get` o `wait`.
- Non importa quali variabili `thread_local` del thread vengono lette o scritte.
- Vi è una garanzia che `get` o `wait` verranno richiamate sul future restituito da `std::async` oppure è accettabile che il task non venga mai eseguito.
- Il codice che utilizza `wait_for` o `wait_until` si assume la responsabilità dello stato di differimento.

Se non vale una di queste condizioni, avete praticamente la garanzia che `std::async` eseguirà lo scheduling del task per un'esecuzione davvero asincrona. Il modo per farlo consiste nel passare `std::launch::async` come primo argomento quando si esegue la chiamata:

```
auto fut =
std::async(std::launch::async,     // lancia f
f);
                                   // in modo asincrono
```

In realtà, il fatto di avere una funzione che si comporta come `std::async`, ma che utilizza automaticamente la politica di lancio `std::launch::async`, è uno strumento comodo; pertanto è positivo che sia facile da scrivere. Ecco la versione C++11:

```
template<typename F, typename...
Ts>
inline
std::future<typename
std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&...            // return future
params)
{                                     // per asincrono
  return                              // chiamata a f(params…)
  std::async(std::launch::async,
      std::forward<F>(f),
      std::forward<Ts>
      (params)...);
}
```

Questa funzione riceve un oggetto richiamabile `f` e zero o più parametri `params` e li inoltra con un perfect-forward (vedi Elemento 25) a `std::async`, passandogli la politica di lancio `std::launch::async`. Come `std::async`, restituisce un `std::future` per il risultato della chiamata di `f` sui `params`. Determinare il tipo di questo risultato è facile, poiché è fornito dal type trait `std::result_of` (per informazioni generali sui type trait, consultate l'Elemento 9).

`reallyAsync` viene utilizzata esattamente come `std::async`:

```
auto fut = reallyAsync(f);        // esegue f in modo asincrono;
                                  // lancia se std::async
                                  // lo lancerebbe
```

In C++14, la possibilità di dedurre il tipo restituito da `reallyAsync` semplifica la dichiarazione della funzione:

```
template<typename F, typename...
Ts>
inline
auto                              // C++14
reallyAsync(F&& f, Ts&&...
params)
{
  return
std::async(std::launch::async,
      std::forward<F>(f),
      std::forward<Ts>
      (params)...);
}
```

Questa versione chiarisce oltre ogni dubbio che `reallyAsync` non fa altro che richiamare `std::async` con la politica di lancio `std::launch::async`.

Argomenti da ricordare

- La politica di lancio di default `std::async` permette l'esecuzione dei task in modo sincrono e asincrono.
- Questa flessibilità porta a una certa incertezza nell'accesso ai `thread_local`, implica che il task potrebbe non venire mai eseguito e influenza la logica del programma per le chiamate `wait` basate su timeout.
- Specificate `std::launch::async` quando è essenziale l'esecuzione asincrona dei task.

Elemento 37 – Rendere gli std::thread non joinable su tutti i percorsi

Ogni oggetto `std::thread` può essere in due diversi stati: *joinable* o *non-joinable*. Uno `std::thread` *joinable* corrisponde a un thread sottostante, che potrebbe essere in esecuzione. Uno `std::thread` corrispondente a un thread sottostante che è bloccato o in attesa di scheduling, è joinable. Gli oggetti

`std::thread` corrispondenti a thread sottostanti che hanno terminato l’esecuzione sono anch’essi considerati joinable.

Uno `std::thread` non-joinable è quello che ci si aspetta: uno `std::thread` che non è joinable. Fra gli oggetti `std::thread` non-joinable vi sono i seguenti.

- **Gli `std::thread` costruiti per default.** Tali `std::thread` non hanno alcuna funzione da eseguire e pertanto non corrispondono a un thread sottostante.
- **Gli oggetti `std::thread` che sono stati spostati.** Il risultato di un’operazione di spostamento è che il thread sottostante cui corrispondeva, eventualmente, uno `std::thread`, ora corrisponde a un altro `std::thread`.
- **Gli `std::thread` che hanno subito una join.** Dopo una join, l’oggetto `std::-thread` non corrisponde più al thread sottostante che ha terminato la propria esecuzione.
- **Gli `std::thread` che hanno subito un `detach`.** Un `detach` recide la connessione fra un oggetto `std::thread` e il thread sottostante cui esso corrisponde.

Un motivo per cui la possibilità di join è importante è che se viene richiamato il distruttore di un oggetto joinable, l’esecuzione del programma termina. Per esempio, supponete di avere una funzione `doWork` che accetta quali parametri una funzione di filtraggio, `filter`, e un valore massimo, `maxVal.` La funzione `doWork` controlla che siano soddisfatte tutte le condizioni necessarie per il proprio calcolo, poi esegue il calcolo con tutti i valori compresi fra `0` e `maxVal,` che passa al filtro. Se l’operazione di filtraggio richiede del tempo e inoltre richiede del tempo anche determinare se le condizioni di `doWork` sono soddisfatte, sarebbe ragionevole far sì che queste due attività vengano svolte in modo concorrente.

La nostra preferenza sarebbe quella di impiegare per questo compito una struttura a task (vedi Elemento 35). Ma supponiamo di voler impostare la priorità del thread che esegue il filtraggio. L’Elemento 35 spiega che ciò richiede l’uso della gestione nativa dei thread, accessibile solo attraverso l’API `std::thread;` l’API basata su task (ovvero i future), non fornisce questa possibilità. Il nostro approccio, pertanto, si baserà sui thread, non sui task.

Potremmo produrre codice come il seguente:

```
constexpr auto tenMillion =
10000000;                        // vedi Elemento 15
                                 // per constexpr

bool
doWork(std::function<bool(int)>  // restituisce il fatto che
filter,
    int maxVal = tenMillion)     // il calcolo è stato
{                                // eseguito; vedi
                                 // Elemento 2 per
                                 // std::function

  std::vector<int> goodVals;     // valori che
                                 // soddisfano il filtro

  std::thread t([&filter, maxVal,
  &goodVals]                     // popola di valori
      {                          // goodVals
        for (auto i = 0; i <=
        maxVal; ++i)
        { if (filter(i))
        goodVals.push_back(i); }
      });

  auto nh = t.native_handle();   // usa l'handle nativo di t
  …                              // per impostare la
                                 // priorità di t

if (conditionsAreSatisfied()) {
  t.join();                      // let t finish
  performComputation(goodVals);
  return true;                   // il calcolo è stato
  }                              // eseguito

  return false;                  // il calcolo non
}                                // è stato eseguito
```

Prima di spiegare perché questo codice è problematico, faccio notare che il valore di inizializzazione di `tenMillion` può essere reso più leggibile in C++14, sfruttando la nuova possibilità di utilizzare un apostrofo come separatore delle cifre:

```
constexpr auto tenMillion = 10'000'000;         // C++14
```

Faccio anche notare che impostare la priorità di `t` *dopo* che ne è iniziata l'esecuzione è un po' come chiudere la proverbiale "porta della stalla" *dopo* che i buoi sono scappati. Sarebbe stato molto meglio avviare `t` in uno stato sospeso (rendendo possibile regolare la sua priorità prima che inizi a svolgere qualsiasi calcolo), ma non volevo distrarvi con il codice necessario per farlo. Se però vi sentite distratti dall'assenza di questa parte del codice, saltate direttamente all'Elemento 39, che mostra come avviare thread sospesi.

Ma torniamo a `doWork`. Se *conditionsAreSatisfied()* restituisce `true`, tutto va bene; se restituisce `false` o lancia un'eccezione, l'oggetto `std::thread t` sarà joinable quando il suo distruttore viene chiamato per chiudere `doWork`. Questo provocherebbe la chiusura del programma.

Potreste chiedervi perché il distruttore di `std::thread` si comporta in questo modo. È perché le altre due opzioni possibili sono perfino peggiori. Eccole.

- **Un `join` implicito.** In questo caso, un distruttore di `std::thread` attenderebbe il completamento del thread asincrono sottostante. Può sembrare ragionevole, ma potrebbe portare ad anomalie prestazionali difficili da risolvere. Per esempio, sarebbe anti-intuitivo che `doWork` attendesse che il suo filtro venisse applicato a tutti valori se *`conditionsAreSatisfied()`* avesse già restituito `false`.
- **Un `detach` implicito.** In questo caso, un distruttore di `std::thread` reciderebbe la connessione fra l'oggetto `std::thread` e il suo thread sottostante. Il thread sottostante continuerebbe a funzionare. Questo non sembra meno ragionevole dell'approccio `join`, ma i problemi di debugging cui può portare sono anche peggiori. In `doWork`, per esempio, `goodVals` è una variabile locale catturata per riferimento. Viene anche modificata all'interno della lambda (tramite la chiamata `push_back`). Supponete, quindi, che mentre la lambda sta operando in modo asincrono,

conditionsAreSatisfied() restituisca `false`. Ciò provocherebbe l'uscita da `doWork` e le sue variabili locali (compresa `goodVals`) verrebbero distrutte. Il suo stack frame verrebbe spazzato via e l'esecuzione del suo thread continuerebbe nel punto di chiamata di `doWork`. Le istruzioni che seguono tale punto di chiamata eseguirebbero, in qualche punto, ulteriori chiamate a funzione e l'ultima di queste chiamate finirebbe per utilizzare una parte della memoria, se non tutta, che un tempo era stata occupata dallo stack frame di `doWork`. Chiamiamo questa funzione `f`. Mentre `f` è in esecuzione, la lambda avviata da `doWork` sarebbe comunque in esecuzione in modo asincrono. Tale lambda potrebbe richiamare `push_back` nell'area dello stack che un tempo era utilizzata da `goodVals`, ma che ora si trova da qualche parte all'interno dello stack frame di `f`. Tale chiamata modificherebbe la memoria che era di `goodVals` e questo significa che, dal punto di vista di `f`, il contenuto della memoria del proprio stack frame potrebbe cambiare spontaneamente! Immaginate quanto possa essere "divertente" eseguire il debugging di *questo* problema.

Il Comitato di Standardizzazione ha deciso che le conseguenze di distruggere un thread joinable erano sufficientemente pericolose da richiederne l'eliminazione (prescrivendo che la distruzione di un thread joinable provochi la terminazione del programma).

Questo dà al programmatore il compito di garantire che, se si usa un oggetto `std::-thread`, questo venga reso non joinable su ogni percorso esterno al campo di visibilità in cui è definito. Ma considerare ogni singolo percorso può essere complesso. Comprende le uscite dal campo di visibilità, ma anche le uscite tramite un `return`, un `continue`, un `break`, un `goto` o un'eccezione. Può trattarsi, davvero, di un gran numero di percorsi.

Ogni volta che si vuole eseguire una qualche azione lungo ciascun percorso all'esterno di un blocco, l'approccio normale consiste nell'inserire tale azione nel distruttore di un oggetto locale. Tali oggetti sono chiamati *oggetti RAII* e le classi da cui provengono sono chiamate *classi RAII (RAII* significa "Resource Acquisition Is Initialization" anche se questa tecnica si occupa di distruzione e non di inizializzazione). Le classi RAII sono comuni nella Libreria Standard. Fra gli esempi vi sono i container STL (il distruttore di ciascun container distrugge il contenuto del container e ne rilascia la memoria), i puntatori smart (gli Elementi da 18 a 20 spiegano che il distruttore di `std::unique_ptr` richiama il proprio cancellatore sull'oggetto cui punta, e i distruttori di `std::shared_ptr` e

std::weak_ptr decrementano il conteggio dei riferimenti), gli oggetti std::fstream (i loro distruttori chiudono i file corrispondenti) e molti altri. E, in ogni caso, non esiste alcuna classe RAII standard per gli oggetti std::thread, forse perché il Comitato di Standardizzazione, avendo rifiutato sia join sia detach come opzioni di default, semplicemente non sapeva cosa dovesse fare una classe di questo tipo.

Fortunatamente non è difficile scriverne una. Per esempio, la seguente classe consente ai chiamanti di specificare se debbano essere chiamati join o detach quando viene distrutto un oggetto ThreadRAII (un oggetto RAII per uno std::thread):

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };    // vedi Elemento 10 per
                                             // info culla classe enum

  ThreadRAII(std::thread&& t, DtorAction a)  // nel distr., esegui
  : action(a), t(std::move(t)) {}            // l'azione a su t

  ~ThreadRAII()
  {                                          // vedi sotto per
    if (t.joinable()) {                      // il test di join

      if (action == DtorAction::join) {
        t.join();
      } else {
        T.detach();
      }
    }
  }

  std::thread& get() { return t; } // vedi sotto
```

```
private:
  DtorAction action;
  std::thread t;
};
```

Credo che il codice precedente sia abbastanza auto-esplicativo, ma le seguenti puntualizzazioni potrebbero essere utili.

- Il costruttore accetta solo rvalue `std::thread`, poiché vogliamo spostare nell'oggetto `ThreadRAII` il `std::thread` passato (ricordate che gli oggetti `std::thread` non sono copiabili).
- L'ordine dei parametri nel costruttore è progettato per essere intuitivo per i chiamati (è più sensato specificare prima lo `std::thread` e poi l'azione del distruttore che non il contrario), ma la lista di inizializzazione dei membri è progettata per corrispondere all'ordine delle dichiarazioni dei dati membro. Tale ordine pone per ultimo l'oggetto `std::thread`. In questa classe, l'ordine non fa alcuna differenza, ma, in generale, è possibile che l'inizializzazione di un dato membro dipenda da quella di un altro e poiché gli oggetti `std::thread` possono avviare una funzione immediatamente dopo essere stati inizializzati, è buona abitudine dichiararli come ultimi in una classe. Ciò garantisce che, nel momento in cui vengono costruiti, tutti i dati membro che li precedono siano già stati inizializzati e possano pertanto essere utilizzati in sicurezza dal thread in esecuzione asincrona che corrisponde al dato membro `std::thread`.
- `ThreadRAII` offre una funzione `get` per fornire l'accesso all'oggetto `std::-thread` sottostante. Questa è analoga alle funzioni `get` offerte dalle classi per puntatori smart per accedere ai puntatori standard sottostanti. Il fatto di fornire `get` evita la necessità per `ThreadRAII` di replicare l'intera interfaccia di `std::-thread` e significa anche che gli oggetti `ThreadRAII` possono essere utilizzati in contesti in cui sono necessari gli oggetti `std::thread`.
- Prima che il distruttore di `ThreadRAII` richiami una funzione membro sull'oggetto `t` di tipo `std::thread`, questo controlla che `t` sia joinable. Questa precauzione è necessaria, poiché richiamare `join` o `detach` su un thread non joinable porta a un comportamento indefinito. È possibile che

un client abbia costruito uno `std::thread`, abbia creato da esso un oggetto `ThreadRAII`, abbia utilizzato `get` per acquisire l'accesso a `t` e poi abbia eseguito uno spostamento da `t` o abbia richiamato `join` o `detach` su di esso. Ognuna di queste azioni renderebbe `t` non-joinable.

Se temete che in questo codice,

```
if (t.joinable()) {

  if (action ==
  DtorAction::join) {
    t.join();
  } else {
    t.detach();
  }
}
```

esista una competizione, poiché fra l'esecuzione di `t.joinable()` e la chiamata a `join` o `detach`, un altro thread potrebbe rendere `t` non-joinable, la vostra intuizione è corretta, ma i vostri timori sono infondati. Un oggetto `std::thread` può cambiare stato da joinable a non-joinable solo attraverso una chiamata a una funzione membro, ovvero `join`, `detach` o un'operazione di spostamento. Nel momento in cui viene richiamato il distruttore di un oggetto `ThreadRAII`, nessun altro thread dovrebbe eseguire chiamate a funzioni membro su tale oggetto. Se vi sono chiamate simultanee, esiste certamente una competizione, ma non all'interno del distruttore, bensì all'interno del codice client che sta tentando di richiamare contemporaneamente sullo stesso oggetto due funzioni membro (il distruttore e un'altra). In generale, le chiamate a funzioni membro simultanee su un unico oggetto sono sicure solo se entrambe sono funzioni membro `const` (vedi Elemento 16).

L'impiego di `ThreadRAII` nel nostro esempio `doWork` avrebbe il seguente aspetto:

```
bool doWork(std::function<bool(int)>
filter,                                  // come prima
      int maxVal = tenMillion)
```

```
{
  std::vector<int> goodVals;       // come prima
  ThreadRAII t(                    // uso dell'oggetto RAII
    std::thread([&filter, maxVal,
    &goodVals]
      {
        for (auto i = 0; i <=
        maxVal; ++i)
          { if (filter(i))
          goodVals.push_back(i);
          }
      }),
      ThreadRAII::DtorAction::join // azione RAII
   );
   auto nh =
   t.get().native_handle();
   …

   if (conditionsAreSatisfied()) {
     t.get().join();
     performComputation(goodVals);
     return true;
   }
   return false;
}
```

In questo caso, abbiamo scelto di eseguire una `join` sul thread in esecuzione asincrona nel distruttore di `ThreadRAII`, poiché, come abbiamo visto in precedenza, l’esecuzione di un `detach` potrebbe portare a un vero incubo di debugging. Abbiamo anche visto in precedenza che l’esecuzione di una `join` potrebbe portare ad anomalie prestazionali (il cui debugging, francamente, potrebbe essere molto spiacevole), ma dovendo scegliere fra comportamento indefinito (derivante dall’uso di `detach`), terminazione del programma (che si avrebbe con l’uso di `std::thread`) o anomalie prestazionali, le anomalie prestazionali sembrano proprio essere il “male minore”.

Purtroppo, l’ Elemento 39 dimostra che l’utilizzo di `ThreadRAII` per eseguire

una `join` sulla distruzione di `std::thread` può portare talvolta non solo ad anomalie prestazionali, ma anche a un blocco del programma. La soluzione "corretta" a questo genere di problemi sarebbe quella di comunicare alla lambda in esecuzione asincrona che non abbiamo più bisogno del suo lavoro e che dovrebbe terminare prematuramente, ma il C++11 non supporta i *thread interrompibili*. Possono essere implementati a mano, ma tale argomento non rientra negli scopi di questo libro.[17]

L'Elemento 17 spiega che, poiché `ThreadRAII` dichiara un distruttore, il compilatore non genererà alcuna operazione di spostamento, ma non vi è alcun motivo per cui gli oggetti `ThreadRAII` non debbano essere spostabili. Se i compilatori dovessero generare queste funzioni, le funzioni farebbero l'operazione corretta; quindi è assolutamente lecito richiedere esplicitamente la loro creazione:

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };          // come prima

  ThreadRAII(std::thread&& t, DtorAction a)        // come prima
  : action(a), t(std::move(t)) {}

  ~ThreadRAII()
  {
    …                                              // come prima
  }

  ThreadRAII(ThreadRAII&&) = default; // supporta
  ThreadRAII& operator=(ThreadRAII&&) = default;   // lo spostamento

  std::thread& get() { return t; }                 // come prima
```

```
private:                          // come prima
  DtorAction action;
  std::thread t;
};
```

Argomenti da ricordare

- Occorre rendere i `std::thread` non-joinable su tutti percorsi.
- Le operazioni di `join` in fase di distruzione possono portare ad anomalie prestazionali di difficile soluzione.
- Le operazioni di `detach` in fase di distruzione possono portare a comportamenti indefiniti di difficile soluzione.
- Dichiarare gli oggetti `std::thread` per ultimi nell'elenco dei dati membro.

Elemento 38 – Attenzione al comportamento variabile del distruttore dell'handle del thread

Come abbiamo visto nell'Elemento 37 un `std::thread` joinable corrisponde a un thread di sistema in esecuzione. Un future per un task non differito (vedi Elemento 36) ha una relazione analoga con un thread di sistema. Per questo motivo, sia gli oggetti `std::thread` sia gli oggetti future possono essere considerati come *handle* per i thread di sistema.

Da questo punto di vista, è interessante notare come i distruttori degli `std::thread` e dei future abbiano comportamenti molto differenti. Come abbiamo visto nell'Elemento 37, la distruzione di uno `std::thread` joinable provoca la chiusura del programma, poiché le due alternative più ovvie (un `join` implicito e un `detach` implicito) sono state considerate scelte peggiori. Al contrario, il distruttore di un future talvolta si comporta come se eseguisse un `join` implicito, talvolta come se eseguisse un `detach` implicito e talvolta come se non eseguisse nessuna di queste due cose. Ma non provoca mai la chiusura del programma. Questo comportamento poco coerente dell'handle del thread merita

un esame più attento.

Inizieremo osservando che un future è un'estremità di un canale di comunicazione tramite il quale un chiamato trasmette un risultato al chiamante.[18] Il chiamato (normalmente in esecuzione in modo sincrono) scrive il risultato del proprio calcolo nel canale di comunicazione (normalmente tramite un oggetto `std::promise`), e il chiamante legge tale risultato utilizzando un future. Potete immaginarvi il meccanismo nel seguente modo, dove la linea tratteggiata mostra il flusso di informazioni dal chiamato al chiamante:

Ma dove viene conservato il risultato del chiamato? Il chiamato potrebbe terminare prima che il chiamante abbia la possibilità di richiamare `get` sul corrispondente future e pertanto, in questo caso, il risultato non può essere memorizzato nello `std::promise` del chiamato. Tale oggetto, essendo locale del chiamato, verrebbe distrutto nel momento in cui terminasse il chiamato.

Il risultato non può neppure essere memorizzato nel future del chiamante, poiché (ma non è l'unico motivo), uno `std::future` potrebbe essere utilizzato per creare uno `std::shared_future` (trasferendo pertanto la proprietà del risultato del chiamato dallo `std::future` allo `std::shared_future`), il quale potrebbe poi essere copiato più volte dopo che lo `std::future` originale è stato distrutto. Dato che non tutti i tipi di risultati possono essere copiati (primi fra tutti i tipi move-only) e dato che il risultato deve sopravvivere almeno quanto l'ultimo future che vi fa riferimento, quale dei potenzialmente numerosi future corrispondenti al chiamato sarà quello che contiene il risultato?

Poiché né gli oggetti associati al chiamato né gli oggetti associati al chiamante sono luoghi adatti per conservare il risultato fornito dal chiamato, questo viene conservato in un'area che è "esterna" a entrambi. Quest'area è chiamata *stato condiviso*. Lo stato condiviso è normalmente rappresentato da un oggetto che si trova sullo heap, ma il suo tipo, la sua interfaccia e la sua implementazione non sono specificati dallo standard. Gli autori della Libreria Standard sono liberi di implementare gli stati condivisi nel modo che desiderano.

Possiamo rappresentare la relazione esistente fra il chiamato, il chiamante e lo stato condiviso nel modo seguente, dove le linee tratteggiate, ancora una volta, rappresentano il flusso delle informazioni:

L'esistenza dello stato condiviso è importante, poiché il comportamento del distruttore di un future, l'argomento di questo Elemento, è determinato dallo stato condiviso associato al future. In particolare valgono le seguenti condizioni.

- **Il distruttore dell'ultimo future che fa riferimento a uno stato condiviso per un task non differito lanciato attraverso `std::async` si blocca** finché il task non si completa. In pratica, il distruttore di questo future esegue un `join` implicito sul thread sul quale è in esecuzione il task asincrono.
- **Il distruttore di tutti gli altri future distrugge semplicemente l'oggetto future.** Per i task in esecuzione asincrona, questo è analogo a un `detach` implicito sul thread sottostante. Per i task differiti per i quali questo è l'ultimo future, significa che il task differito non funzionerà mai.

Queste regole sembrano più complicate di quanto non siano in realtà. Esiste un semplice comportamento "normale", con un'unica piccola eccezione. Il comportamento normale è che il distruttore di un future distrugge l'oggetto future. Questo è tutto. Non esegue il `join` con nulla, non esegue il `detach` da nulla. Non lancia nulla. Non fa altro che distruggere i dati membro del future. In realtà, fa anche molte altre cose. Decrementa il conteggio dei riferimenti all'interno dello stato condiviso che è manipolato sia dai future che vi fanno riferimento sia dal `std::promise` del chiamato. Questo conteggio dei riferimenti consente alla libreria di sapere quando lo stato condiviso può essere distrutto. Per informazioni generali sul conteggio dei riferimenti, consultate l'Elemento 19).

L'eccezione a questo comportamento normale si ha solo per un future per il quale valgono tutte le seguenti condizioni.

- **Fa riferimento a uno stato condiviso che è stato creato a causa di una chiamata a `std::async`.**
- **La politica di lancio del task è `std::launch::async`** (vedi Elemento 36); o perché questa è stata scelta dal sistema runtime oppure perché è stata specificata nella chiamata a `std::async`.

- **Questo è l'ultimo future che fa riferimento allo stato condiviso.** Per gli `std::future`, questa sarà la situazione tipica. Per gli `std::shared_futures`, se vi sono altri `std::shared_future` che fanno riferimento allo stesso stato condiviso del future che sta per essere distrutto, il future da distruggere seguirà il comportamento normale (ovvero distrugge semplicemente i suoi dati membro).

Solo quando tutte queste condizioni sono esaudite il distruttore di un future esibisce il comportamento speciale e tale comportamento consiste nel bloccarsi finché non termina il task a esecuzione asincrona. In termini pratici, questo equivale a un `join` implicito con il thread che esegue il task creato da `std::async`.

È molto comune sentire riassumere questo comportamento normale del distruttore del future come "I future di `std::async` si bloccano nei loro distruttori". Come approssimazione è corretta, ma, talvolta è necessario essere più precisi. A questo punto conoscerete tutta la verità, in tutta la sua meraviglia.

Naturalmente potreste avere un concetto molto differente di "meraviglia". Potrebbe essere del tipo "Mi meraviglio che non ci sia una speciale regola per gli stati condivisi per i task non differiti che vengono lanciati da `std::async`". È una domanda ragionevole. Per quanto ne sappiamo, il Comitato di Standardizzazione voleva evitare i problemi associati a un `detach` implicito (vedi Elemento 37), ma non voleva adottare una politica così radicale come la terminazione obbligatoria del programma (come ha fatto per gli `std::thread` joinable, vedi, ancora, l'Elemento 37), pertanto il compromesso trovato è stato un `join` implicito. La decisione è stata un po' controversa e vi sono state profonde discussioni sul fatto di abbandonare questo comportamento per il C++14. Alla fine, non è stata eseguita alcuna modifica e dunque il comportamento dei distruttori per i future è coerente sia in C++11 sia in C++14.

L'API per i future non offre alcun modo per determinare se un future fa riferimento a uno stato condiviso derivante da una chiamata a `std::async`; pertanto, dato un oggetto future arbitrario, non è possibile sapere se si bloccherà nel proprio distruttore in attesa che termini un task a esecuzione asincrona. Ciò ha alcune implicazioni interessanti:

```
// questo container può bloccarsi nel suo distr., poiché uno o più
// future contenuti può far riferimento a uno
```

```
stato condiviso per un
// task non differito lanciato tramite std::async
std::vector<std::future<void>> futs;          // vedi Elemento 39 per info
                                              // su std::future<void>

class Widget {                                // gli oggetti Widget devono
public:                                       // bloccarsi nei loro distruttori
…

private:
  std::shared_future<double> fut;
};
```

Naturalmente, se avete un modo per sapere che un determinato future *non* soddisfa le condizioni che provocano il comportamento speciale del distruttore (magari a causa della logica del programma), avete la garanzia che tale future non si bloccherà nel suo distruttore. Per esempio, solo gli stati condivisi che derivano dalle chiamate a `std::async` richiedono il comportamento speciale, ma vi sono altri modi con cui possono essere creati gli stati condivisi. Uno è l'uso di `std::packaged_task`. Un oggetto `std::packaged_task` prepara una funzione (o un altro oggetto chiamante) per l'esecuzione asincrona, includendola in modo che il suo risultato venga inserito in uno stato condiviso. Un future che facesse riferimento a tale stato condiviso potrebbe pertanto essere ottenuto tramite la funzione `get_future` di `std::packaged_task`:

```
int calcValue();                  // funzione da eseguire

std::packaged_task<int()>         // ingloba calcValue per poterla
  pt(calcValue);                  // eseguire in modo asincrono
auto fut = pt.get_future();       // get future per pt
```

A questo punto, sappiamo che il future `fut` non fa riferimento a uno stato condiviso creato da una chiamata a `std::async`, pertanto il suo distruttore si

comporterà normalmente.

Una volta creato, il `std::packaged_task` pt può essere eseguito su un thread. Potrebbe anche essere eseguito attraverso una chiamata `std::async`, ma, se volete seguire un task utilizzando `std::async`, non vi sono molti motivi per creare uno `std::packaged_task`, poiché `std::async` fa tutto ciò che fa `std::packaged_task` prima di eseguire lo scheduling dell'esecuzione del task.

Gli `std::packaged_task` non sono copiabili e pertanto, quando pt viene passato al costruttore di `std::thread`, deve essere convertito in un rvalue (tramite `std::move`, vedi Elemento 23):

```
std::thread t(std::move(pt));     // esegue pt su t
```

Questo esempio getta luce sul comportamento normale del distruttore di future, ma è più facile vedere il tutto se le istruzioni vengono inserite in un blocco:

```
{                                 // inizio blocco

  std::packaged_task<int()>
    pt(calcValue);

  auto fut = pt.get_future();
  std::thread t(std::move(pt));

…                                 // vedi sotto

}                                 // fine blocco
```

Il codice più interessante, qui, è il "…" che segue la creazione dell'oggetto `t` di tipo `std::thread` che precede la fine del blocco. È interessante per ciò che accade a `t` all'interno della regione "…". Sostanzialmente vi sono tre possibilità.

- **A `t` non accade nulla.** In questo caso, `t` sarà joinable alla fine del campo di visibilità (scope). Ciò provocherà la chiusura del programma (vedi Elemento 37).

- **Viene eseguita una `join` su `t`.** In questo caso, non vi sarebbe alcuna necessità che `fut` si bloccasse nel proprio distruttore, poiché la `join` è già presente nel codice chiamante.
- **Viene eseguito un `detach` su `t`.** In questo caso, non vi sarebbe alcuna necessità che `fut` eseguisse il `detach` nel proprio distruttore, poiché l'ha già fatto il codice chiamante.

In altre parole, quando avete un future che corrisponde a uno stato condiviso prodotto da uno `std::packaged_task`, in genere non vi è alcuna necessità di adottare una particolare politica di distruzione, poiché la decisione fra terminazione, `join` o `detach` verrà presa nel codice che manipola lo `std::thread` su cui è tipicamente in esecuzione lo `std::packaged_task`.

Argomenti da ricordare

- I distruttori di future normalmente si occupano semplicemente di distruggere i dati membro del future.
- L'ultimo future che fa riferimento a uno stato condiviso per un task non differito lanciato tramite `std::async` si blocca finché non è completato tutto il task.

Elemento 39 – Considerate l'uso di future void per la comunicazione di eventi "one-shot"

Talvolta è utile che un task comunichi a un secondo task in funzionamento asincrono che si è verificato un determinato evento, perché il secondo task non può procedere finché non si è svolto tale evento. Magari è stata inizializzata una struttura dati, è stata completata una fase del calcolo o è stato rilevato uno specifico valore da un sensore. In questo caso, qual è il modo migliore per svolgere questo genere di comunicazione fra thread?

Un approccio ovvio consiste nell'utilizzare una variabile di condizione (*condvar*). Se chiamiamo il task che rileva la condizione con il nome di *task di*

rilevamento e il task che reagisce alla condizione con il nome di *task di reazione*, la strategia è semplice: il task di reazione attende la variabile di condizione e il task di rilevamento notifica tale variabile di condizione quando si verifica l’evento. Dato

```
std::condition_variable cv;        // condvar per l'evento

std::mutex m;                      // mutex da usare con cv
```

il codice del task di rilevamento è estremamente semplice:

```
…                                  // rileva l'evento

cv.notify_one();                   // lo comunica al task di
                                   reazione
```

Se la notifica avesse bisogno di raggiungere più task di reazione, sarebbe appropriato sostituire `notify_one` con `notify_all`, ma, per il momento, supporremo che vi sia un solo task di reazione.

Il codice del task di reazione è un po’ più complesso, poiché prima di chiamare `wait` sulla condvar, deve bloccare un mutex attraverso un oggetto `std::unique_lock`. Il bloccaggio di un mutex in attesa di una variabile di condizione è tipico delle librerie per thread. La necessità di bloccare il mutex attraverso un oggetto `std::unique_lock` fa semplicemente parte dell’API del C++11. Ecco l’approccio teorico:

```
…                                  // si prepara a reagire

{                                  // apre la sezione critica

  std::unique_lock<std::mutex>     // blocca il mutex
  1k(m);

  cv.wait(lk);                     // attende la notifica;
                                   // questo non è errato!
```

```
  …                                   // reagisce all'evento
                                      // (m è bloccato)

}                                     // close crit. section;
                                      // sblocca n tramite il distr. di
                                      lk

…                                     // continua la reazione
                                      // (ora m è sbloccato)
```

Il primo problema di questo approccio è un'impressione generale di *malfunzionamento*: anche se il codice funziona, non sembra del tutto giusto. In questo caso, il problema deriva dalla necessità di utilizzare un mutex. I mutex vengono utilizzati per controllare l'accesso ai dati condivisi, ma è assolutamente possibile che i task di rilevamento e di reazione non abbiano alcuna necessità di tale mediazione. Per esempio, il task di rilevamento potrebbe essere responsabile dell'inizializzazione di una struttura dati globale, da passare poi al task di reazione che dovrà utilizzarla. Se il task di rilevamento non accede mai alla struttura dati dopo averla inizializzata e se il task di reazione non accede mai alla struttura dati prima che il task di rilevamento indichi che è pronta, i due task si troveranno perfettamente separati nel flusso di funzionamento del programma. Non vi è alcuna necessità di utilizzare un mutex. Il fatto che l'approccio a condvar ne richieda uno getta un'ombra sospetta sulla struttura del programma.

Ma, anche ignorando questo dubbio, vi sono altri due problemi ai quali è assolutamente necessario dedicare attenzione.

- **Se il task di rilevamento notifica la condvar prima che il task di reazione lanci `wait`, quest'ultimo si bloccherà.** Perché la notifica di una condvar risvegli un altro task, quest'altro task deve essere in attesa della condvar. Se il task di rilevamento esegue la notifica prima che il task di reazione esegua il `wait`, il task di reazione si perderà la notifica e resterà in attesa per sempre.
- **L'istruzione `wait` non tiene in considerazione i risvegli spuri.** È un fatto della vita nelle API per thread (in molti linguaggi, non solo in C++) che il codice in attesa di una variabile di condizione possa essere risvegliato

anche se la condvar non è stata notificata. Tali risvegli sono chiamati i *risvegli spuri*. Un codice ben realizzato li gestisce verificando che la condizione attesa si sia effettivamente verificata, e lo può fare come prima azione dopo il risveglio. L'API per condvar C++ rende questa operazione estremamente semplice, poiché permette il passaggio a wait di una lambda (o di un altro oggetto funzione) che controlli la condizione in questione. In pratica, la chiamata a `wait` nel task di reazione potrebbe essere scritta nel seguente modo:

```
cv.wait(lk,[]{ return il fatto che l'evento si sia verificato;
});
```

Per sfruttare questa possibilità, è necessario che il task di reazione sia in grado di determinare se la condizione che sta attendendo è verificata. Ma, nella situazione che stiamo considerando, la condizione attesa è il verificarsi di un evento che dovrà essere riconosciuto dal thread di rilevamento. Il thread di reazione potrebbe non avere alcun modo per determinare se l'evento che si sta verificando si è effettivamente svolto. Questo è il motivo per cui attende una variabile di condizione!

Vi sono molte situazioni in cui il fatto che i task comunichino attraverso una condvar rappresenta una buona soluzione per il problema, ma questa non sembra essere così fortunata.

Per molti sviluppatori, una soluzione potrebbe essere un flag booleano condiviso. Il flag è inizialmente `false`. Quando il thread di rilevamento riconosce l'evento che sta ricercando, alza il flag:

```
std::atomic<bool> flag(false);    // flag condiviso; vedi
                                  // Elemento 40 per std::atomic

…                                 // rileva l'evento
flag = true;                      // lo comunica al task di
                                  reazione
```

Da parte sua, il thread di reazione non fa altro che interrogare il flag. Quando vede che il flag è true, sa che l'evento che sta attendendo si è verificato:

```
…                                  // si prepara a reagire
while (!flag);                     // attende l'evento
…                                  // reagisce all'evento
```

Questo approccio non soffre di nessuno dei difetti dell'approccio a condvar. Non vi è la necessità di impiegare un mutex, nessun problema se il task di rilevamento alza il flag prima che il task di reazione inizi a interrogarlo e nessuna sensibilità ai risvegli spuri. Sembra un approccio perfettamente funzionante!

Meno positivo è il costo della continua interrogazione che si svolge nel task di reazione. Mentre il task è in attesa dell'attivazione del flag, tale task è sostanzialmente bloccato, anche se è in esecuzione. Per questo motivo, occupa un thread hardware che potrebbe essere utilizzato da un altro task, subisce il costo della commutazione di contesto ogni volta che parte o termina il proprio time-slice e potrebbe mantenere in esecuzione un core che altrimenti potrebbe essere chiuso per risparmiare energia. Un task davvero bloccato non farebbe nessuna di queste cose. Questo è un vantaggio dell'approccio basato su condvar, poiché un task in una chiamata `wait` è davvero bloccato.

È pratica comune combinare entrambe queste soluzioni: a condvar e a flag. Un flag indica se l'evento in questione si è verificato, ma l'accesso al flag è sincronizzato da un mutex. Poiché il mutex impedisce l'accesso concorrente al flag, non vi è, come spiega l'Elemento 40, nessuna necessità che il flag sia `std::atomic`. Basta un semplice `bool`. Il task di rilevamento avrà pertanto il seguente aspetto:

```
std::condition_variable cv;        // come prima
std::mutex m;

bool flag(false);                  // non std::atomic

…                                  // rileva l'evento

{
  std::lock_guard<std::mutex>      // blocca m tramite il costr. di
  g(m);                            g
```

```
  flag = true;                    // lo comunica al task di
                                  reazione
                                  // (parte 1)

}                                 // sblocca n tramite il distr. Di
                                  g
cv.notify_one();                  // lo comunica al task di
                                  reazione
                                  // (parte 2)
```

Ed ecco anche il task di reazione:

```
…                                 // si prepara a reagire

{                                 // come prima
  std::unique_lock<std::mutex>    // come prima
  lk(m);

  cv.wait(lk, [] { return flag;   // usa una lambda per evitare
  });
                                  // i risvegli spuri

  …                               // reagisce all'evento
                                  // (m è bloccato)

}

…                                 // continua la reazione
                                  // (ora m è sbloccato)
```

Questo approccio evita i problemi di cui abbiamo parlato. Funziona indipendentemente dal fatto che il task di reazione sia in attesa prima che il task di rilevamento invii la notifica, funziona in presenza di risvegli spuri e non richiede continue interrogazioni. Rimane però un dubbio, poiché il task di rilevamento comunica con il task di reazione in un modo piuttosto curioso. La notifica della variabile di condizione dice al task di reazione che l’evento che sta

attendendo si è probabilmente verificato, ma il task di reazione deve controllare il flag per sincerarsene. L'impostazione del flag conferma al task di reazione che l'evento si è verificato, ma il task di rilevamento deve ancora notificare la variabile di condizione, in modo che il task di reazione venga risvegliato per controllare il flag. L'approccio funziona, ma non sembra del tutto lineare.

Un'alternativa consiste nell'evitare le variabili di condizione, i mutex e i flag, facendo sì che il task di reazione attenda un future impostato dal task di rilevamento. Questa non sembra essere una buona idea. Dopotutto, l'Elemento 38 spiega che un future rappresenta l'estremità ricevente di un canale di comunicazione da un chiamato a un chiamante (tipicamente asincrono) e qui non esiste una relazione chiamato-chiamante fra i task di rilevamento e di reazione. Tuttavia, l'Elemento 38 ci dice anche che un canale di comunicazione in cui l'estremità di trasmissione è una `std::promise` e l'estremità di ricezione è un future può essere utilizzato per altre cose oltre che per una classica comunicazione chiamato-chiamante. Tale canale di comunicazione può essere utilizzato in ogni situazione in cui occorra trasmettere informazioni da un punto del programma a un altro. In questo caso, lo utilizzeremo per trasmettere informazioni dal task di rilevamento al task di reazione e l'informazione che invieremo sarà proprio il fatto che l'evento atteso si è verificato.

La struttura è semplice. Il task di rilevamento ha un oggetto `std::promise` (ovvero l'estremità scrivente del canale di comunicazione) e il task di reazione ha un corrispondente future. Quando il task di rilevamento vede che l'evento che sta controllando si è verificato, *imposta* `std::promise` (ovvero scrive nel canale di comunicazione). Nel frattempo, il task di reazione attende il suo future. Questo `wait` blocca il task di reazione finché `std::promise` non è stato impostato.

Ora, sia `std::promise` sia i future (ovvero `std::future` e `std::shared_future`) sono template che richiedono un parametro per il tipo. Tale parametro indica il tipo dei dati che verranno trasmessi attraverso il canale di comunicazione. Tuttavia, nel nostro caso non vi sono dati da inviare. L'unica cosa che interessa il task di reazione è il fatto che il suo future sia stato impostato. Ciò di cui abbiamo bisogno per i template di `std:: promise` e del future è un tipo che indichi che non devono essere inviati dati attraverso il canale di comunicazione. Questo tipo è `void`. Il task di rilevamento utilizzerà pertanto uno `std::promise<void>` e il task di reazione uno `std::future<void>` o uno `std::shared_future<void>`. Il task di rilevamento imposterà il proprio

`std::promise<void>` nel momento in cui si verifica l'evento in questione e il task di reazione sarà in `wait` sul suo future. Anche se il task di reazione non riceverà dati dal task di rilevamento, il canale di comunicazione permetterà al task di reazione di sapere quando il task di rilevamento ha "scritto" il proprio dato `void` richiamando `set_value` sul proprio `std::promise`.

Pertanto, dato

```
std::promise<void> p;             // promise per
                                  // il canale di comunicazione
```

il codice del task di rilevamento è banale,

```
…                                 // rileva l'evento

p.set_value();                    // lo comunica al task di
                                  reazione
```

e il codice del task di reazione è altrettanto semplice:

```
…                                 // si prepara a reagire

p.get_future().wait();            // attende il future
                                  // corrispondente a p

…                                 // reagisce all'evento
```

Come l'approccio che utilizza un flag, questa soluzione non richiede alcun mutex, funziona indipendentemente dal fatto che il task di rilevamento imposti il proprio `std::promise` prima che il task di reazione entri in attesa con `wait` ed è immune ai risvegli spuri (solo le variabili di condizione sono sensibili a questo problema). Come l'approccio basato su condvar, il task di reazione è davvero bloccato dopo la chiamata a `wait` e dunque non consuma alcuna risorsa di sistema mentre è in attesa. Tutto perfetto, vero?

Non esattamente. Certamente, un approccio basato su future evita questi problemi, ma vi sono altri rischi di cui preoccuparsi. Per esempio, l'Elemento 38

spiega che tra un `std::promise` e un future si trova uno stato condiviso e gli stati condivisi, in genere, sono allocati in modo dinamico. Si dovrebbe pertanto presumere che questa soluzione incorra nei costi di operazioni di allocazione e deallocazione sullo heap.

Una cosa ancora più importante: uno `std::promise` può essere impostato una sola volta. Il canale di comunicazione fra uno `std::promise` e un future è un meccanismo *one-shot*: non può essere utilizzato ripetutamente. Questa è un'importante differenza rispetto alle versioni condvar e flag, le quali potevano essere utilizzate per comunicare più volte (una condvar può essere notificata ripetutamente e un flag può sempre essere cancellato e poi reimpostato).

Il limite di "una volta sola" non è però così grave come potrebbe sembrare. Supponete di voler creare un thread di sistema in uno stato sospeso. In pratica vorreste eliminare tutto il sovraccarico associato alla creazione del thread, in modo che quando siete pronti per eseguire qualcosa sul thread, venga evitata la normale latenza dovuta alla creazione del thread. Oppure potreste voler creare un thread sospeso, in modo da poterlo configurare prima di utilizzarlo. Tale configurazione potrebbe comprendere cose come l'impostazione della sua priorità o l'affinità ai core. L'API per la concorrenza del C++ non offre alcun modo per fare queste cose, ma gli oggetti `std::thread` offrono la funzione membro `native_handle`, il cui risultato ha lo scopo di dare accesso al'API di gestione dei thread della piattaforma (normalmente thread POSIX o Windows). L'API di basso livello spesso consente di configurare queste caratteristiche dei thread, ovvero la priorità e l'affinità.

Supponendo di voler sospendere un thread una sola volta (dopo la creazione ma prima che sia in esecuzione la sua funzione thread) una soluzione che utilizzi un future `void` rappresenta una scelta ragionevole. Ecco l'essenza di questa tecnica:

```
std::promise<void> p;

void react();                      // funzione del task di reazione

void detect()                      // funzione del task di
                                   rilevamento
{
  std::thread t([]                 // crea il thread
```

```
        {
          p.get_future().wait();  // sospende t finché
          react();                // il future è impostato
      });
…                                 // qui, t è sospeso
                                  // prima della chiamata di
                                  reazione

p.set_value();                    // rilascia t (e pertanto
                                  // avvia la reazione)

  …                               // lavoro aggiuntivo

  t.join();                       // rende t non-joinable
}                                 // (vedi Elemento 37)
```

Poiché è importante che `t` divenga non-joinable su tutti i percorsi tranne `detect`, sembrerebbe consigliabile impiegare una classe `RAII` come la `ThreadRAII` presentata nell'Elemento 37. Potrebbe venire in mente del codice avente il seguente aspetto:

```
void detect()
{
  ThreadRAII tr(                  // uso dell'oggetto RAII
    std::thread([]
        {
          p.get_future().wait();
          react();
        }),
    ThreadRAII::DtorAction::join  // rischioso! (vedi sotto)
  );

  …                               // il thread in tr
                                  // qui è sospeso
```

```
p.set_value();                    // thread non sospeso
                                  // in tr
  …
}
```

Questo codice sembra più sicuro di quanto non sia in realtà. Il problema è che se nella prima regione “...” (quella con il commento “ il thread in `tr` qui è sospeso “) viene emessa un’eccezione, `set_value` non verrà mai chiamata su `p`. Ciò significa che la chiamata `wait` all’interno della lambda non verrà mai eseguita. Questo, a sua volta, significa che il thread che esegue la lambda non si concluderà mai e questo è un problema, poiché l’oggetto RAII `tr` è stato configurato per eseguire una `join` su tale thread nel distruttore di `tr`. In altre parole, se viene emessa un’eccezione per la prima regione di codice “…”, questa funzione si bloccherà, poiché il distruttore di `tr` non completerà mai il proprio funzionamento. Vi sono dei modi per risolvere questo problema, ma li lascerò come esercizio al lettore.[19] Qui vorrei mostrare come il codice originale (ovvero quello che non utilizza `ThreadRAII)` possa essere esteso per sospendersi e poi uscire dalla sospensione non solo una volta, ma più volte. È una semplice generalizzazione, poiché la soluzione consiste nell’utilizzare degli `std::shared_future` al posto degli `std::future` nel codice di `react.` Una volta che si sa che la funzione membro share di `std::shared_future` trasferisce la proprietà del proprio stato condiviso all’oggetto `std::future's share` prodotto da `share,` il codice praticamente si scrive da solo. L’unico dettaglio è il fatto che ogni thread di reazione ha bisogno della propria copia dello `std::shared_future` che fa riferimento allo stato condiviso, in modo che lo `std::shared_future` ottenuto da `share` venga catturato per valore dalle lambda in esecuzione sui thread di reazione:

```
std::promise<void> p;             // come prima

void detect()                     // ora per più
{                                 // task di reazione

  auto sf =
  p.get_future().share();         // il tipo di sf è
                                  // std::shared_future<void>
```

```
  std::vector<std::thread> vt;     // container per
                                   // i thread di reazione

  for (int i = 0; i <
  threadsToRun; ++i) {
    vt.emplace_back([sf]{          // attendi la copia
    sf.wait();
          react(); });             // locale di sf; vedi
}                                  // Elemento 42 per info
                                   // su emplace_back

…                                  // rileva i blocchi se
                                   // questo codice "…" viene
                                   lanciato!

  p.set_value();                   // rilascia tutti i thread
  …

  for (auto& t : vt) {             // rende tutti i thread
    t.join();                      // non-joinable; vedi Elemento 2
  }                                // per info su "auto&"
}
```

Il fatto che una soluzione che utilizza il future possa ottenere questo effetto è degno di nota e questo è il motivo per cui dovreste considerarne l'impiego per tutte le comunicazioni di tipo one-shot.

Argomenti da ricordare

- Per le semplici comunicazioni fra eventi, le soluzioni basate su condvar richiedono un mutex superfluo, impongono dei vincoli sul progresso relativo dei task di rilevamento di reazione e richiedono che i task di reazione verifichino che l'evento si sia effettivamente svolto.
- Le soluzioni che impiegano un flag non hanno questo problema, ma sono basate su attività di interrogazione, senza bloccaggio del task.

- È possibile utilizzare insieme una condvar e un flag, ma il meccanismo di comunicazione risultante è per certi versi eccessivo.
- L'uso di `std::promise` e dei future risolve questi problemi, ma l'approccio utilizza della memoria sullo heap per gli stati condivisi e il suo impiego è limitato alle comunicazioni one-shot.

Elemento 40 – Usare std::atomic per la concorrenza e volatile per la memoria speciale

Povera `volatile`. Così incompresa. Non dovrebbe nemmeno trovarsi in questo capitolo, poiché non ha nulla a che fare con la programmazione concorrente. Ma, in altri linguaggi di programmazione (per esempio Java e C#), è utile proprio per queste attività e anche in C++ alcuni compilatori hanno dotato `volatile` di elementi semantici che la rendono applicabile al software concorrente (ma solo quando viene compilata con questi compilatori). Pertanto vale la pena di parlare di `volatile` in un capitolo sulla concorrenza, quanto meno per fugare la confusione che la circonda.

La funzionalità del C++ con cui alcuni programmatori confondono talvolta `volatile` (e che decisamente fa parte di questo capitolo) è il template `std::atomic`. Le istanziazioni di questo template (per esempio `std::atomic<int>`, `std::atomic<bool>`, `std::atomic<Widget*>` e così via) offrono operazioni che hanno la garanzia di essere viste come atomiche dagli altri thread.

Una volta che un oggetto `std::atomic` è stato costruito, le operazioni su di esso si comportano come se fossero all'interno di una sezione critica protetta da un mutex, ma le operazioni sono generalmente implementate utilizzando speciali istruzioni macchina e sono più efficienti di quelle impiegate nel caso di un mutex.

Considerate il seguente codice che utilizza `std::atomic`:

```
std::atomic<int> ai(0);            // inizializza ai a 0
```

```
ai = 10;                        // imposta in modo atomico ai a
                                10

std::cout << ai;                // legge in modo atomico il
                                valore di ai

++ai;                           // incrementa in modo atomico ai
                                a 11

--ai;                           // decrementa in modo atomico ai
                                a 10
```

Durante l'esecuzione di queste istruzioni, gli altri thread che leggono `ai` potrebbero vedere solo i valori 0, 10 o 11. Non sono possibili altri valori (supponendo, naturalmente, che questo sia l'unico thread che modifica `ai`).

Due aspetti di questo esempio sono degni di nota. Innanzitutto, nell'istruzione `std:: cout << ai;`, il fatto che `ai` sia `std::atomic` garantisce solo che la lettura di `ai` sia atomica. Non esiste alcuna garanzia che l'intera istruzione venga elaborata in modo atomico. Tra il momento in cui viene letto il valore di `ai` e il momento in cui `operator<<` viene richiamato per scriverlo sullo standard output, un altro thread potrebbe aver modificato il valore di `ai`. Questo non ha alcun effetto sul comportamento dell'istruzione, poiché `operator<<` usa un parametro passato per valore per l'**`int`** da produrre in output (il valore prodotto sarà pertanto quello che è stato letto da `ai`), ma è importante comprendere che in questa istruzione è atomica solamente la lettura di `ai`.

Il secondo aspetto degno di nota di questo esempio è il comportamento delle ultime due istruzioni, l'incremento e il decremento di `ai`. Si tratta di operazioni leggi-modificascrivi (RMW, ovvero read-modify-write) e pertanto vengono eseguite in modo atomico. Questa è una delle caratteristiche più importanti dei tipi `std::atomic`: una volta che un oggetto `std::atomic` è stato costruito, tutte le funzioni membro al suo interno, incluse quelle che comprendono operazioni RMW, hanno la garanzia di essere viste come atomiche dagli altri thread.

Al contrario, il codice corrispondente che utilizza `volatile` non garantisce praticamente niente in un contesto multi-thread:

```
volatile int vi(0);             // inizializza vi a 0
```

```
vi = 10;                          // imposta vi a 10

std::cout << vi;                  // legge il valore di vi

++vi;                             // incrementa vi a 11

--vi;                             // decrementa vi a 10
```

Durante l’esecuzione di questo codice, se altri thread leggono il valore di `vi`, potrebbero trovarvi qualsiasi cosa, per esempio -12, 68, 4090727, letteralmente qualsiasi cosa! Tale codice avrebbe pertanto un comportamento indefinito, poiché queste istruzioni modificano `vi` in modo che, se contemporaneamente altri thread stanno leggendo `vi`, vi sono operazioni di lettura e scrittura sulla memoria che non sono né std::atomic né protette da un mutex, e questa è esattamente una competizione sui dati.

Come esempio concreto del modo in cui il comportamento di `std::atomic` e dei `volatile` può differire in un programma multi-thread, considerate un semplice contatore di ciascun tipo che viene incrementato da più thread. Inizializzeremo entrambi a 0:

```
std::atomic<int> ac(0);           // "contatore atomico"

volatile int vc(0);               // "contatore volatile"
```

Incrementeremo poi ciascun contatore una volta in due thread in esecuzione simultaneamente:

```
    /*----- Thread 1 -----*/        /*------- Thread 2 -------*/

             ++ac;                              ++ac;
             ++vc;                              ++vc;
```

Al termine di entrambi i thread, il valore di `ac` (ovvero il valore dello `std::atomic`) deve essere 2, poiché ciascun incremento si verifica come

un’operazione indivisibile. Il valore di `vc`, al contrario, non è necessariamente 2, poiché i suoi incrementi possono non verificarsi in modo atomico. Ogni incremento è costituito dalla lettura del valore di `vc`, dall’incremento del valore letto e dalla scrittura del risultato di nuovo in `vc`. Ma non abbiamo alcuna garanzia che queste tre operazioni procedano in modo atomico per gli oggetti `volatile` e pertanto è possibile che le operazioni dei due incrementi di `vc` si svolgano nel seguente modo.

1. Il thread 1 legge il valore di `vc`, che è 0.
2. Il thread 2 legge il valore di `vc`, che è ancora 0.
3. Il thread 1 incrementa il valore 0 che ha letto portandolo a 1, poi scrive tale valore in `vc`.
4. Il thread 2 incrementa il valore 0 che ha letto portandolo a 1, poi scrive tale valore in `vc`.

Il valore finale di `vc` sarà pertanto 1, anche se tale valore è stato incrementato per due volte.

Questo non è l’unico risultato possibile. Il valore finale di `vc` è, in generale, imprevedibile, poiché `vc` è coinvolto in una competizione sui dati e il “decreto” dello standard sul fatto che la competizione sui dati provoca un comportamento indefinito significa che i compilatori possono generare codice che può fare letteralmente qualsiasi cosa. I compilatori non usano questa tecnica di proposito, naturalmente. Piuttosto svolgono ottimizzazioni che sarebbero valide nei programmi esenti da competizioni e queste ottimizzazioni provocano comportamenti inattesi e imprevedibili nei programmi in cui sussistono competizioni sui dati.

L’uso di operazioni RMW non è l’unica situazione in cui gli `std::atomic` rappresentano una soluzione di successo della concorrenza e i `volatile` falliscono l’obiettivo. Supponete che un task calcoli un valore importante richiesto da un secondo task. Quando il primo task ha calcolato il valore, deve comunicarlo al secondo task. L’Elemento 39 spiega che un modo con cui il primo task comunica al secondo task la disponibilità del valore desiderato è l’impiego di uno `std::atomic<bool>` . Il codice presente nel task che calcola il valore avrebbe all’incirca il seguente aspetto:

```
std::atomic<bool>
valAvailable(false);
```

```
auto imptValue =
computeImportantValue();          // calcola il valore

valAvailable = true;              // dice agli altri task
                                  // che è disponibile
```

Quando noi leggiamo questo codice, capiamo che è fondamentale che l'assegnamento a `imptValue` si svolga prima dell'assegnamento `valAvailable`, ma tutto ciò che vedranno i compilatori è una coppia di assegnamenti a variabili fra loro indipendenti. Come regola generale, i compilatori hanno il permesso di cambiare l'ordine degli assegnamenti non correlati. In pratica, data questa sequenza di assegnamenti (dove `a`, `b`, `x` e `y` corrispondono a variabili indipendenti),

```
a = b;
x = y;
```

i compilatori possono disporre nel seguente modo:

```
x = y;
a = b;
```

Anche se i compilatori non eseguono questo riordino delle operazioni, l'hardware sottostante potrebbe farlo (o potrebbe far sembrare agli altri core di averlo fatto), poiché per qualche strano motivo ciò farebbe eseguire il codice in modo più veloce.

Tuttavia, l'uso dei `std::atomic` impone delle restrizioni sul modo in cui il codice può essere riordinato e una di queste restrizioni è il fatto che nessuna riga che, nel codice sorgente, *precede* la scrittura di una variabile `std::atomic` possa svolgersi (o far sembrare agli altri core di svolgersi) *dopo* tale scrittura.[20] Ciò significa che nel nostro codice,

```
auto imptValue =
computeImportantValue();          // calcola il valore
valAvailable = true;              // dice agli altri task
                                  // che è disponibile
```

non solo i compilatori devono mantenere l'ordine degli assegnamenti a `imptValue` e a `valAvailable`, ma devono anche generare del codice che garantisca che anche l'hardware sottostante faccia la stessa cosa. Come risultato, la dichiarazione di `valAvailable` come `std::atomic` garantisce che il nostro requisito di ordinamento critico (tutti i thread devono vedere che `imptValue` cambia prima di `valAvailable`) sia rispettato.

La dichiarazione di `valAvailable` come `volatile` non impone le stesse restrizioni sul riordinamento del codice:

```
volatile bool
valAvailable(false);

auto imptValue =
computeImportantValue();

valAvailable = true;            // altri thread potrebbero vedere
                                questo assegnamento
                                // prima di quello a imptValue!
```

Qui, i compilatori possono scambiare l'ordine degli assegnamenti a `imptValue` e `valAvailable` e, anche se non lo fanno, potrebbero non riuscire a generare il codice macchina, che impedirebbe all'hardware sottostante di far sì che il codice su altri core veda i cambiamenti applicati a `valAvailable` prima di quelli applicati a `imptValue`.

Questi due problemi (nessuna garanzia dell'atomicità dell'operazione e restrizioni insufficienti sul riordinamento del codice) spiegano perché `volatile` non è utile per la programmazione concorrente, ma non spiega a che cosa serve. In sostanza, `volatile` dice ai compilatori che hanno a che fare con memoria che non si comporta in modo normale.

La memoria "normale" ha caratteristiche "normali": se si scrive un valore in un'area di memoria, tale valore rimane tale finché qualcuno non lo modifica. Pertanto, se esiste un normale `int`,

```
int x;
```

e un compilatore trova la seguente sequenza di operazioni su di esso,

```
auto y = x;                          // legge x
y = x;                               // legge ancora x
```

sa di poter ottimizzare il codice generato eliminando l'assegnamento a `y`, poiché è ridondante, data l'inizializzazione precedente.

Un'altra caratteristica della memoria "normale": se si scrive un valore in un'area di memoria, non lo si legge mai e poi si scrive nuovamente su tale area di memoria, la prima scrittura viene eliminata, poiché non è mai stata usata. Pertanto, date queste due istruzioni adiacenti,

```
x = 10;                              // scrive su x
x = 20;                              // scrive ancora su x
```

i compilatori sono in grado di eliminare la prima. Questo significa che se nel codice sorgente abbiamo la seguente forma,

```
auto y = x;                          // legge x
y = x;                               // legge nuovamente x
x = 10;                              // scrive su x
x = 20;                              // scrive ancora su x
```

i compilatori possono trattarla come se fosse scritta nel seguente modo:

```
auto y = x;                          // legge x
x = 20;                              // scrive su x
```

Forse vi domandate chi mai scriverebbe del codice che svolge questo genere di letture ridondanti e di scritture superflue (chiamate tecnicamente *redundant loads* e *dead stores*); la risposta è che forse nessun programmatore lo farebbe di proposito. Tuttavia, dopo che i compilatori hanno preso del codice sorgente con un aspetto razionale e vi svolgono istanziazioni di template, inserimenti inline, varie altre operazioni comuni di riordinamento, può capitare che il risultato contenga letture ridondanti e scritture superflue, di cui i compilatori possono sbarazzarsi.

Tali ottimizzazioni sono valide solo se la memoria si comporta in modo normale.

La memoria “speciale” non lo fa. Il tipo probabilmente più comune di memoria speciale è quella utilizzata per le operazioni di *I/O mappate in memoria*. Queste aree di memoria comunicano con delle periferiche, per esempio dei sensori esterni, delle stampanti, delle porte di rete e così via, invece di contare su normali operazioni di lettura o scrittura in memoria (tipicamente la memoria RAM). In questo contesto, considerate nuovamente il codice contenente letture apparentemente ridondanti:

```
auto y = x;                         // legge x
y = x;                              // legge nuovamente x
```

Se `x` corrisponde a, poniamo, il valore rilevato da un sensore di temperatura, la seconda lettura di `x` non è affatto ridondante, perché nel frattempo la temperatura può essere cambiata.

Si ha una situazione analoga per operazioni di scrittura apparentemente superflue. Nel seguente codice, per esempio,

```
x = 10;                             // scrive su x
x = 20;                             // scrive di nuovo su x
```

se `x` corrisponde alla porta di controllo di un trasmettitore radio, può essere che il codice emetta dei comandi per la radio e che il valore 10 corrisponda a un comando differente dal valore 20. Eliminando il primo assegnamento si cambierebbe la sequenza di comandi inviati all’apparato radio.

Per dire ai compilatori che abbiamo a che fare con della memoria speciale, si utilizza `volatile`. Per un compilatore, `volatile` ha il seguente significato: “non eseguire alcuna operazione di ottimizzazione su questa memoria”. Pertanto, se `x` corrisponde a un’area di memoria speciale, dovrebbe essere dichiarata `volatile`:

```
volatile int x;
```

Considerate l’effetto che ha sulla nostra sequenza di codice originale:

```
auto y = x;                         // legge x
y = x;                              // legge nuovamente x (non
                                    ottimizzare)
```

```
x = 10;                    // scrive su x (non ottimizzare)
x = 20;                    // scrive di nuovo su x
```

Questo è esattamente ciò che vogliamo se `x` è mappata in memoria (o è stata mappata a un'area di memoria condivisa fra più processi e così via).

Piccola domanda! In quest'ultimo frammento di codice, qual è il tipo di `y`: è `int` oppure `volatile int`?[21]

Il fatto che le letture apparentemente ridondanti e le scritture apparentemente superflue debbano essere mantenute quando si ha a che fare con aree di memoria speciali spiega, a proposito, perché gli `std::atomic` non sono adatti a questo genere di lavoro. I compilatori potrebbero eliminare queste operazioni ridondanti sugli `std::atomic`. Il codice non è scritto allo stesso modo impiegato normalmente per i `volatile`, ma, se sorvoliamo per un momento su questo aspetto e ci concentriamo su ciò che i compilatori possono fare, possiamo vedere che, teoricamente, i compilatori potrebbero prendere questo codice,

```
std::atomic<int> x;
auto y = x;                // teoricamente legge x (vedi sotto)
y = x;                     // teoricamente legge nuovamente x (vedi sotto)

x = 10;                    // scrive su x
x = 20;                    // scrive di nuovo su x
```

e ottimizzarlo nel seguente modo:

```
auto y = x;                // teoricamente legge x (vedi sotto)
x = 20;                    // scrive su x
```

Per la memoria speciale, questo è un comportamento decisamente inaccettabile.

Ora, com'è giusto, nessuna di queste due istruzioni può essere compilata se `x` è `std:: atomic`:

```
auto y = x;                         // errore!
y = x;                              // errore!
```

Questo perché le operazioni di copia per `std::atomic` vengono cancellate (vedi Elemento 11). E vi è un buon motivo. Considerate cosa accadrebbe se l'inizializzazione di `y` con `x` passasse la compilazione. Poiché `x` è `std::atomic`, il tipo di `y` verrebbe anch'esso dedotto come `std::atomic` (vedi Elemento 2). Ho detto in precedenza che una delle migliori caratteristiche degli `std::atomic` è il fatto che tutte le loro operazioni sono atomiche, ma perché la costruzione per copia di `y` da `x` sia atomica, i compilatori dovranno generare del codice per leggere `x` e scrivere `y` in un'unica operazione atomica. L'hardware, generalmente, non può farlo, pertanto la costruzione per copia non è supportata per i tipi `std::atomic`. L'assegnamento per copia deve essere eliminato per la stessa ragione, e questo è il motivo per cui l'assegnamento da `x` a `y` non viene compilato (le operazioni di spostamento non sono dichiarate esplicitamente in `std::atomic`, e pertanto, per le regole delle funzioni speciali generate dal compilatore descritte nell'Elemento 17, `std::atomic` non offre né la costruzione per spostamento né l'assegnamento per spostamento).

È possibile inserire in `y` il valore di `x`, ma richiede l'uso delle funzioni membro `load` e `store` di `std::atomic`. La funzione membro `load` legge in modo atomico il valore di uno `std::atomic`, mentre la funzione membro `store` lo scrive in modo atomico. Per inizializzare `y` con `x` per poi inserire il valore di `x` in `y`, il codice deve essere scritto nel seguente modo:

```
std::atomic<int> y(x.load());      // legge x

y.store(x.load());                 // legge nuovamente x
```

Questo codice viene compilato, ma il fatto che la lettura di `x` (tramite `x.load()`) sia una chiamata a funzione distinta rispetto all'inizializzazione o alla memorizzazione in `y` chiarisce che non vi è alcun motivo per aspettarsi che le istruzioni vengano eseguite come un'unica operazione atomica.

Dato questo codice, i compilatori potrebbero "ottimizzarlo" memorizzando il valore di `x` in un registro invece di leggerlo per due volte:

```
register = x.load();               // legge x in register
```

```
std::atomic<int> y(register);          // inizializza y con il valore di
                                       register

y.store(register);                     // memorizza in y il valore di
                                       register
```

Il risultato, come potete vedere, è che `x` viene letta una sola volta, e questo è proprio il genere di ottimizzazioni da evitare quando si ha a che fare con la memoria speciale (l'ottimizzazione non è permessa per le variabili `volatile`).

La situazione, a questo punto, dovrebbe essere chiara.

- **`std::atomic`** è utile per la programmazione concorrente, ma non per accedere alla memoria speciale.
- **`volatile`** è utile per accedere alla memoria speciale, ma non per la programmazione concorrente.

Poiché `std::atomic` e `volatile` hanno scopi differenti, possono anche essere utilizzate insieme:

```
volatile std::atomic<int> vai;    // le operazioni su vai sono
                                  // atomiche e non possono
                                  // essere ottimizzate
```

Questo può essere utile se `vai` corrisponde a un'area di memoria mappata in operazioni di I/O il cui accesso può avvenire da più thread.

Come nota finale, alcuni sviluppatori preferiscono utilizzare le funzioni membro `load` e `store` di `std::atomic` anche quando non sarebbe obbligatorio, poiché rendono esplicito nel codice sorgente che le variabili coinvolte non sono "normali". Enfatizzare questo fatto non è affatto irragionevole. L'accesso a uno `std::atomic` è tipicamente molto più lento rispetto a un accesso a un non-`std::atomic` e abbiamo già visto che l'uso di `std::atomic` impedisce ai compilatori di svolgere determinate attività di ridisposizione del codice che, altrimenti, sarebbero permesse. La chiamata delle operazioni di `load` e `store` degli `std::atomic` può pertanto aiutare a identificare potenziali problemi di scalabilità. Dal punto di vista della correttezza, il fatto di *non trovare* una

chiamata a `store` su una variabile destinata a comunicare informazioni ad altri thread (per esempio un flag che indica la disponibilità dei dati) potrebbe significare che la variabile non è stata dichiarata `std::atomic`, mentre avrebbe dovuto esserlo.

Si tratta, per certi versi, di una questione di stile e non coinvolge direttamente la scelta fra `std::atomic` e `volatile`.

Argomenti da ricordare

- `std::atomic` è rivolta ai dati cui si fa accesso da più thread senza utilizzare mutex. È uno strumento per la scrittura di software concorrente.
- `volatile` è invece rivolta alla memoria in cui le operazioni di lettura e scrittura non dovrebbero essere ottimizzate. È uno strumento per operare su aree di memoria speciali.

---

[15]. Non sempre il sistema operativo esiste. Alcuni sistemi embedded non lo usano.

[16]. Questa è una semplificazione. Quello che conta non è il future su cui viene richiamato get o wait, è lo stato condiviso cui fa riferimento il future (la relazione esistente fra i future e gli stati condivisi è trattata nell'Elemento 38). Poiché std::future supporta lo spostamento e può essere anche utilizzato per costruire degli `std::shared_future` e poiché gli `std::shared_future` possono essere copiati, l'oggetto future che fa riferimento allo stato condiviso che deriva dalla chiamata a `std::async` cui viene passata f sarà probabilmente differente da quello restituito da `std::async`. Questo però è un dettaglio tanto complesso quanto insignificante, pertanto è pratica comune dire semplicemente che si richiama get o wait sul future restituito da `std::async`.

[17]. Potete trovare un'ottima trattazione nel testo di Anthony Williams, *C++ Concurrency in Action* (Manning Publications, 2012).

[18]. L'Elemento 39 spiega che il tipo di canale di comunicazione associato a un future può essere impiegato anche per altri scopi. Rimanendo in questo Elemento, tuttavia, considereremo solo il suo uso quale meccanismo con il quale

un chiamato invia il proprio risultato a un chiamante.

[19]. Una risorsa da cui potete iniziare la ricerca su questo argomento è il mio post sul blog del 24 dicembre 2013 in *The View From Aristeia*, “ThreadRAII + Thread Suspension = Trouble?”.

[20]. Questo vale solo per gli `std::atomic` che usano la *coerenza sequenziale*, che è sia il modello di default sia, anche, l’unico modello di coerenza per gli oggetti `std::atomic` che usano la sintassi presentata in questo libro. Il C++11 supporta anche modelli di coerenza con regole più flessibili di ridisposizione del codice. Tali modelli deboli (o *rilassati*) consentono di creare software che può operare più velocemente su alcune architetture hardware, ma l’uso di questi modelli genera software che è molto più difficile da far funzionare, da comprendere e da correggere. Capita frequentemente che sorgano errori subdoli nel codice che utilizza oggetti atomici rilassati, difficilmente rilevabili anche dagli esperti. Pertanto, se possibile, utilizzate sempre la coerenza sequenziale.

[21]. Il tipo di `y` viene dedotto automaticamente, pertanto utilizza le regole descritte nell’Elemento 2. Queste regole stabiliscono che, per la dichiarazione di tipi non riferimento e non puntatore (e questo è il caso di `y`), i qualificatori `const` e `volatile` vengano eliminati. Il tipo di `y`, pertanto, è semplicemente `int`. Questo significa che le letture ridondanti e le scritture superflue di `y` possono essere eliminate. Nell’esempio, i compilatori devono svolgere sia l’inizializzazione sia l’assegnamento a `y`, poiché `x` è volatile, pertanto la seconda lettura di `x` potrebbe fornire un valore differente dalla prima.

8

Tecniche varie

Per ogni tecnica generale o funzionalità del C++, ci sono circostanze in cui è ragionevole utilizzarla, ma anche precise circostanze in cui è meglio evitarla. Descrivere quando ha senso utilizzare una determinata tecnica generale o funzionalità, normalmente, è piuttosto semplice, ma questo capitolo si occupa di due eccezioni. La tecnica generale è il passaggio per valore; la funzionalità generale è l'emplacement. La decisione sul loro impiego è influenzata da molti fattori e pertanto il miglior consiglio che posso offrirvi è quello di *valutarne* l'uso. Ciononostante, si tratta in entrambi i casi di elementi importanti nella programmazione C++ moderna e i due Elementi che seguono forniscono tutte le informazioni necessarie per aiutarvi a determinare se il loro uso è appropriato per il vostro software.

Elemento 41 – Considerare il passaggio per valore per i parametri la cui copia è "poco costosa" e che pertanto vengono sempre copiati

Alcuni parametri di funzioni hanno proprio lo scopo di essere copiati.[22] Per esempio, una funzione membro `addName` potrebbe copiare il proprio parametro in un container privato.

Per considerazioni di efficienza, tale funzione dovrebbe copiare gli argomenti lvalue e spostare gli argomenti rvalue:

```
class Widget {
public:
  void addName(const std::string&           // prende l’lvalue;
  newName)
  { names.push_back(newName); }             // lo copia

  void addName(std::string&& newName)       // prende l'rvalue;
  {
  names.push_back(std::move(newName));      // lo sposta;
  }
…                                           // vedi Elemento 25 per
                                            l'uso
                                            // di std::move

private:
  std::vector<std::string> names;
};
```

Questo codice funziona, ma richiede la scrittura di due funzioni che svolgono sostanzialmente la stessa cosa. Questo irrita un po’: due funzioni da dichiarare, due funzioni da implementare, due funzioni da documentare, due funzioni da correggere. Non andiamo bene!

Inoltre, vi saranno due funzioni nel codice oggetto, un fattore da valutare quando si è preoccupati dei problemi di “ingombro” del programma. In questo caso, entrambe le funzioni verranno probabilmente messe inline e questo probabilmente eliminerà ogni problema di eccesso dimensionale delle due funzioni, ma, se il compilatore non mette inline queste due funzioni ogni volta, vi ritroverete con due funzioni nel codice oggetto.

Un approccio alternativo consiste nel rendere `addName` un template di funzione che riceve un riferimento universale (vedi Elemento 24):

```
class Widget {
public:
  template<typename T>                // prende gli lvalue
  void addName(T&& newName)           // e gli rvalue;
  {                                   // copia gli lvalue,
    names.push_back(std::forward<T>   // sposta gli rvalue;
    (newName));
  }                                   // vedi Elemento 25
                                      // sull'uso
…                                     // di std::forward
};
```

Ciò riduce il codice sorgente di cui preoccuparsi, ma l'impiego dei riferimenti universali genera nuove complicazioni. Trattandosi di un template, l'implementazione di `addName` deve normalmente finire in un file header. Può generare parecchie funzioni nel codice oggetto, poiché non solo si istanzia in modo differente per lvalue e rvalue, ma si istanzia in modo differente anche per le `std::string` e per i tipi che sono convertibili in `std::string` (vedi Elemento 25). Contemporaneamente, vi sono tipi di argomenti che non possono essere passati per riferimento universale (vedi Elemento 30) e se i client passano tipi di argomenti impropri, i messaggi di errore prodotti dal compilatore possono essere davvero spaventosi (vedi Elemento 27).

Come sarebbe bello se vi fosse un modo per scrivere funzioni come `addName` in modo che gli lvalue venissero copiati, gli rvalue venissero spostati, vi fosse una sola funzione di cui occuparsi (nel codice sorgente e anche nel codice oggetto) evitando nel contempo tutte le idiosincrasie dei riferimenti universali! E si dà il caso che questo modo esista. Tutto ciò che dovete fare è agire in deroga a una delle prime regole che avete probabilmente imparato come programmatori C++. Tale regola diceva di evitare di passare per valore gli oggetti dei tipi definiti dall'utente. Ma per parametri come `newName` in funzioni come `addName`, il passaggio per valore può essere una strategia perfettamente ragionevole.

Prima di vedere perché il passaggio per valore può essere adatto a `newName` e `addName`, vediamo come verrebbe implementato:

```
class Widget {
public:
```

```
  void addName(std::string newName)    // prende l'lvalue o
  {
  names.push_back(std::move(newName)); // l'rvalue; lo sposta
  }
  …
};
```

L’unica parte non ovvia di questo codice è l’applicazione di `std::move` al parametro `newName`. Tipicamente, `std::move` viene utilizzata con i riferimenti rvalue, ma, in questo caso, sappiamo che (1) `newName` è un oggetto completamente indipendente da qualsiasi cosa abbia passato il chiamante, pertanto ogni modifica a `newName` non influenzerà il chiamante e (2), questo è l’utilizzo finale di `newName`, e pertanto anche se verrà spostato, ciò non avrà alcun impatto sul resto della funzione.

Il fatto che vi sia una sola funzione `addName` spiega come evitiamo la duplicazione del codice, sia nel codice sorgente sia nel codice oggetto. Non utilizziamo un riferimento universale, quindi questo approccio non causa il proliferare di file header, situazioni non considerate o messaggi di errore confusi. Ma vogliamo parlare dell’efficienza di questa soluzione? Stiamo effettuando un passaggio per valore. Non è costoso?

In C++98, normalmente, era proprio così. Indipendentemente da ciò che passava il chiamante, il parametro `newName` sarebbe stato creato tramite *costruzione per copia.* Al contrario, in C++11, `addName` verrebbe costruito per copia solo per gli lvalue. Per gli rvalue, verrebbe *costruito per spostamento.* Ecco qui:

```
Widget w;


…
std::string name("Bart");

w.addName(name);                  // richiama addName con un lvalue


…
w.addName(name + "Jenne");        // richiama addName con un rvalue
                                  // (vedi sotto)
```

Nella prima chiamata ad `addName` (dove viene passato `name`), il parametro `newName` viene inizializzato con un lvalue. `newName` viene pertanto costruito per copia, così come avverrebbe in C++98. Nella seconda chiamata, `newName` viene inizializzata con l'oggetto `std::string` risultante da una chiamata a `operator+` per `std::string` (ovvero l'operazione di append). Tale oggetto è un rvalue e pertanto `newName` viene costruito per spostamento.

Pertanto, gli lvalue vengono copiati e gli rvalue vengono invece spostati, esattamente come vogliamo. Bello, no?

È bello, ma bisogna tenere in considerazione alcuni dettagli. Ricapitoliamo le tre versioni di `addName` che abbiamo considerato:

```
class Widget {                         // Approccio 1:
public:                                // overload per
  void addName(const std::string&      // lvalue ed
  newName)
  { names.push_back(newName); }        // rvalue

  void addName(std::string&& newName)
  {
  names.push_back(std::move(newName));
  }
  …

private:
  std::vector<std::string> names;
};

class Widget {                         // Approccio 2:
public:                                // riferimento
  template<typename T>                 // universale
  void addName(T&& newName)
  { names.push_back(std::forward<T>
  (newName)); }
…
};
```

```
class Widget {                              // Approccio 3:
public:                                     // passaggio per valore
  void addName(std::string newName)
  {
  names.push_back(std::move(newName));
  }
…
};
```

Chiamerò le prime due versioni con il nome di “approcci per riferimento”, poiché entrambe sono basate sul passaggio dei parametri per riferimento.

Ecco le due situazioni di chiamata che abbiamo esaminato:

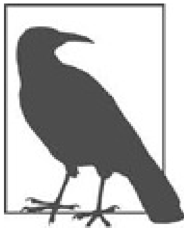
```
Widget w;
…
std::string name("Bart");
w.addName(name);                       // passa un lvalue
…
w.addName(name + "Jenne");             // passa un rvalue
```

Ora considerate il costo, in termini di operazioni di copia e di spostamento, dell’aggiunta di un nome a un Widget per le due situazioni di chiamata e ognuna delle tre implementazioni di `addName` di cui abbiamo parlato. Il conteggio ignora sostanzialmente la possibilità che i compilatori possano ottimizzare le operazioni di copia e spostamento, poiché tali ottimizzazioni dipendono dal contesto e sono nelle mani del compilatore; poi, tutto sommato, non cambiano l’essenza dell’analisi.

- **Overloading:** indipendentemente dal fatto che venga passato un lvalue o un rvalue, l’argomento del chiamante è associato a un riferimento chiamato `newName`. Questo non costa nulla, in termini di operazioni di copia e spostamento. Nell’overloading per lvalue, `newName` viene copiato in `Widget::names`. Nell’overloading per rvalue, viene spostato. Riepilogo dei costi: una copia per gli lvalue, uno spostamento per gli rvalue.
- **Uso di un riferimento universale:** come con l’overloading, l’argomento del chiamante viene associato al riferimento `newName`. Questa è un’operazione senza costi. A causa dell’uso di `std::forward`, gli

argomenti `std::string` lvalue vengono copiati in `Widget::names`, mentre gli argomenti `std::string` rvalue vengono spostati. Il riepilogo dei costi per gli argomenti `std::string` è lo stesso dell'overloading: una copia per gli lvalue, uno spostamento per gli rvalue. L' Elemento 25 spiega che se un chiamante passa un argomento di un tipo diverso da `std::string`, questo verrà inoltrato a un costruttore di `std::string` e, se tutto va bene, questo potrebbe non richiedere alcuna operazione di copia o spostamento di `std::string`. Le funzioni che accettano riferimenti universali possono pertanto essere particolarmente efficienti. Tuttavia, ciò non influenza l'analisi di questo Elemento, pertanto semplifichiamo le cose supponendo che i chiamanti passino sempre argomenti `std::string`.

- **Passaggio per valore:** indipendentemente dal fatto che venga passato un lvalue o un rvalue, il parametro `newName` deve essere costruito. Se è stato passato un lvalue, questa operazione ha il costo di una costruzione per copia. Se viene passato un rvalue, il costo è quello di una costruzione per spostamento. Nel corpo della funzione, `newName` viene spostato incondizionatamente in `Widget::names`. Il riepilogo dei costi è quindi di una copia più uno spostamento per gli lvalue e due spostamenti per gli rvalue. Rispetto agli approcci per riferimento, questo richiede uno spostamento in più sia per gli lvalue sia per gli rvalue.



Ora rileggete un'altra volta il titolo di questo Elemento: *Considerare il passaggio per valore per i parametri la cui copia è "poco costosa" e che pertanto vengono sempre copiati.*

Se abbiamo usato queste parole, ci sarà un motivo. Quattro motivi, in realtà.

1. È il caso di *valutare* (niente di più) il passaggio per valore. D'accordo, richiede la scrittura di una sola funzione. D'accordo, genera una sola funzione nel codice oggetto. D'accordo, evita i problemi associati ai riferimenti universali. Ma presenta un costo più elevato rispetto alle alternative e, come vedremo più avanti, in alcuni casi, occorre considerare anche altri costi.

2. Considerate il passaggio per valore solo per i *parametri copiabili.* I

parametri che non rientrano in questa categoria devono avere tipi move-only, poiché se non sono copiabili, ma la funzione esegue sempre una copia, la copia deve essere creata tramite il costruttore per spostamento.[23] Ricordate che il vantaggio del passaggio per valore rispetto all'overloading è il fatto che con il passaggio per valore deve essere scritta una sola funzione. Ma per i tipi move-only, non vi è alcuna necessità di fornire un overload per gli argomenti lvalue, poiché la copia di lvalue richiede la chiamata al costruttore per copia e il costruttore per copia per i tipi move-only è disabilitato. Ciò significa che devono essere supportati solo gli argomenti rvalue e, in tal caso, la soluzione di "overloading" richiede un solo overload: quello che accetta un riferimento rvalue.

Considerate una classe con un dato membro `std::unique_ptr<std::string>` e il relativo setter. `std::unique_ptr` è di tipo move-only, pertanto l'approccio a overloading per il suo setter è costituito da un'unica funzione:

```
class Widget {
public:
…
   void setPtr(std::unique_ptr<std::string>&& ptr)
   { p = std::move(ptr); }
private:
   std::unique_ptr<std::string> p;
};
```

Un chiamante potrebbe utilizzarlo nel seguente modo:

```
Widget w;
…
w.setPtr(std::make_unique<std::string>("Modern C++"));
```

Qui l'rvalue `std::unique_ptr<std::string>` restituito da `std::make_unique` (vedi Elemento 21) viene passato a `setPtr` per riferimento rvalue. Qui viene spostato nel dato membro `p`. Il costo totale è di uno spostamento.

Se `setPtr` dovesse prendere il proprio parametro per valore,

```
class Widget {
public:
  …
  void setPtr(std::unique_ptr<std::string> ptr)
  { p = std::move(ptr); }
  …
};
```

la stessa chiamata costruirebbe il parametro `ptr` per spostamento e `ptr` verrebbe poi assegnato per spostamento al dato membro `p`. Il costo totale sarebbe pertanto di due spostamenti, il doppio rispetto all'approccio a overloading.

3. Vale la pena di considerare il passaggio per valore solo per i parametri che sono economici da spostare. Quando gli spostamenti sono economici, il costo di uno spostamento aggiuntivo può essere accettabile, quando invece non lo sono, l'esecuzione di uno spostamento inutile è analoga all'esecuzione di una copia inutile e l'importanza di evitare operazioni di copia inutili è proprio ciò che ha portato alla regola del C++98 che consiglia di evitare i passaggi per valore!

4. Dovreste considerare il passaggio per valore solo per i parametri che vengono *sempre copiati*. Per capire perché quest'aspetto è importante, supponete che prima di copiare il proprio parametro nel container `names`, `addName` controlli se il nuovo nome è troppo breve o troppo lungo. Se lo è, la richiesta di aggiungere il nome verrà ignorata. Un'implementazione a passaggio per valore potrebbe essere scritta nel seguente modo:

```
class Widget {
public:
  void addName(std::string newName)
  {
    if ((newName.length() >= minLen) && (newName.length() <=
      maxLen))
      {
        names.push_back(std::move(newName));
```

```
        }
    }
```

Questa funzione incorre nel costo della costruzione e distruzione di `newName`, anche se nulla è stato aggiunto a `names`. Questo è un prezzo che gli approcci a passaggio per riferimento non sono costretti a pagare.

Anche quando si ha a che fare con una funzione che svolge una copia incondizionata su un tipo copiabile che sarebbe economico spostare, vi sono casi in cui il passaggio per valore può non essere appropriato. Questo perché una funzione può copiare un parametro in due modi: tramite *costruzione* (ovvero costruzione per copia o costruzione per spostamento) e tramite *assegnamento* (ovvero tramite assegnamento per copia o assegnamento per spostamento). `addName` utilizza la costruzione: il suo parametro `newName` viene passato a `vector::push_back` e, all'interno di tale funzione, `newName` viene costruito per copia in un nuovo elemento creato alla fine di `std::vector`. Per le funzioni che usano la costruzione per copiare il proprio parametro, l'analisi che abbiamo visto in precedenza è completa: utilizzando il passaggio per valore si incorre nel costo di uno spostamento aggiuntivo per argomenti lvalue ed rvalue.

Quando un parametro viene copiato utilizzando l'assegnamento, la situazione è più complessa. Supponete, per esempio, di avere una classe che rappresenta delle password. Poiché le password possono essere modificate, prevediamo anche una funzione setter, `changeTo`. Utilizzando una strategia a passaggio per valore, potremmo implementare `Password` nel seguente modo:

```
class Password {
public:
  explicit Password(std::string pwd)      // passaggio per valore
  : text(std::move(pwd)) {}               // costruisce text

  void changeTo(std::string newPwd)       // passaggio per valore
  { text = std::move(newPwd); }           // assegna text

  …

private:
```

```
  std::string text;               // testo della password
};
```

Il fatto di memorizzare la password come testo in chiaro solleverà le ire di ogni responsabile della sicurezza che si rispetti, ma ignoriamo questo aspetto e consideriamo il seguente codice:

```
std::string initPwd("Supercalifragilisticexpialidocious");

Password p(initPwd);
```

Nessuna sorpresa: `p.text` viene costruita con la password fornita e l'uso del passaggio per valore nel costruttore incorre nel costo di una costruzione per spostamento di una `std::string` che non sarebbe necessaria se venissero impiegati l'overloading o il perfect-forward. Tutto bene.

Un utente di questo programma potrebbe non gradire troppo la password scelta, perché, in molti dizionari, si trova anche la parola "Supercalifragilisticexpialidocious". Potrebbe pertanto prevedere azioni che portano all'esecuzione di codice equivalente al seguente:

```
std::string newPassword = "Beware the Jabberwock";

p.changeTo(newPassword);
```

Che la nuova password sia migliore della vecchia, è molto discutibile, ma questo è un problema dell'utente. Il nostro problema, invece, è il fatto che l'utilizzo di `changeTo` nell'assegnamento per copiare il parametro `newPwd` fa sì che, probabilmente, la strategia di passaggio per valore utilizzata dalla funzione genererà un notevole incremento dei costi.

L'argomento passato a `changeTo` è un lvalue (`newPassword`), pertanto quando viene costruito il parametro `newPwd`, viene richiamato il costruttore per copia di `std::string`. Questo costruttore alloca la memoria per contenere la nuova password. `newPwd` viene poi assegnato per spostamento a `text`, il che causa la deallocazione della memoria precedentemente allocata per `text`. All'interno di `changeTo` vi sono pertanto due azioni di gestione della memoria dinamica: una per allocare la memoria per la nuova password e una per deallocare la memoria per la vecchia password.

Ma, in questo caso, la vecchia password ("Supercalifragilisticexpialidocious") è più lunga della nuova ("Beware the Jabberwock"), pertanto non vi è alcuna

necessità di allocare o deallocare nulla. Se fosse stato utilizzato l'approccio a overloading, probabilmente non si verificherebbe proprio nulla:

```
class Password {
public:
  …

  void changeTo(const std::string& newPwd)    // l'overload
  {                                            // per lvalue

    text = newPwd;                             // può riutilizzare la memoria di text
                                               // se text.capacity() >= newPwd.size()
  }

  …
private:
  std::string text;                            // come sopra
};
```

In questa situazione, il costo del passaggio per valore comprende anche un'operazione aggiuntiva di allocazione e deallocazione della memoria, costi che superano probabilmente di uno o più ordini di grandezza quelli di un'operazione di spostamento di una `std::string`.

Un aspetto interessante: se la vecchia password fosse stata più breve della nuova, sarebbe stato praticamente impossibile evitare una coppia di operazioni allocazione-deallocazione durante l'assegnamento e, in tal caso, il passaggio per valore avrebbe avuto all'incirca la stessa velocità del passaggio per riferimento. Il costo della copia del parametro basato su assegnamento può pertanto dipendere dal valore degli oggetti che partecipano all'assegnamento stesso! Questo tipo di analisi si applica a ogni tipo di parametri che contenga dei valori situati nella memoria allocata dinamicamente. Non tutti i tipi sono adatti, ma molti, fra cui `std::string` e `std::vector`, sì.

Questo potenziale incremento dei costi si applica, in genere, solo quando

vengono passati argomenti lvalue, poiché la necessità di svolgere operazioni di allocazione e deallocazione della memoria si verifica tipicamente solo quando vengono svolte vere operazioni di copia (e non di spostamento). Per gli argomenti che sono rvalue, quasi sempre sono sufficienti gli spostamenti.

Il risultato è che il costo aggiuntivo del passaggio per valore per le funzioni che copiano un parametro utilizzando un assegnamento dipende dal tipo passato, dal fatto che gli argomenti siano lvalue o rvalue, dal fatto che il tipo utilizzi memoria allocata dinamicamente e, in questo caso, dall'implementazione degli operatori di assegnamento di tale tipo e dalla probabilità che la memoria associata alla destinazione dell'assegnamento sia grande almeno quanto la memoria associata all'origine dell'assegnamento. Per `std::string`, dipende anche dal fatto che l'implementazione utilizzi l'ottimizzazione SSO (small string optimization, vedi Elemento 29) e, in questo caso, dal fatto che i valori assegnati entrino nel buffer SSO.

Pertanto, come ho detto, quando i parametri vengono copiati per assegnamento, l'analisi dei costi del passaggio per valore è complessa. Normalmente, l'approccio più pratico consiste nell'adottare un atteggiamento del tipo "colpevole fino a prova contraria", ogni volta che si usa l'overloading o i riferimenti universali invece del passaggio per valore, a meno che sia dimostrato che il passaggio per valore produca codice sensibilmente più efficiente per il tipo di parametro in uso.

Ora, per software che deve essere il più veloce possibile, il passaggio per valore può essere una strategia da scartare, poiché può essere importante evitare anche gli spostamenti economici. Inoltre, non sempre è chiaro quanti spostamenti si svolgeranno. Nell'esempio di `Widget::addName`, il passaggio per valore richiede un'unica operazione aggiuntiva di spostamento, ma supponete che `Widget::addName` chiami `Widget::validateName` e che anche questa funzione utilizzi il passaggio per valore (presumibilmente ha un motivo per copiare sempre il suo parametro, ovvero per memorizzarlo in una struttura dati di tutti i valori che convalida) e supponete che `validateName` chiami una terza funzione ancora con un passaggio per valore...

Potete vedere dove stiamo andando con questo ragionamento. Quando vi sono catene di chiamate a funzioni, ognuna delle quali impiega il passaggio per valore poiché "costa solo un economicissimo spostamento in più", il costo dell'intera catena di chiamate potrebbe diventare davvero sensibile. Utilizzando il

passaggio dei parametri per riferimento, le catene di chiamate non incorreranno in questo tipo di sovraccarico da accumulo.

Un altro problema non è, invece, legato alle prestazioni, ma è sempre il caso di tenerlo in considerazione: il passaggio per valore, a differenza del passaggio per riferimento, è suscettibile al *problema dello slicing.* Si tratta di un vecchio problema del C++98 e dunque non è il caso di parlarne troppo, ma se avete una funzione che è progettata per accettare un parametro il cui tipo è una classe base o *un qualsiasi tipo derivato da essa,* non vorrete dichiarare un parametro di passaggio per valore di tale tipo, poiché “elimineresteˮ le caratteristiche della classe derivata di qualsiasi oggetto derivato che sia stato passato:

```
class Widget { … };                  // classe base

class SpecialWidget: public
Widget { … };                        // classe derivata

void processWidget(Widget w);        // funzione per ogni tipo di
                                     // Widget,
                                     // compresi i tipi derivati;
…                                    // soffre del problema dello
                                     // slicing

SpecialWidget sw;

…

processWidget(sw);                   // processWidget vede un
                                     // Widget, non uno SpecialWidget!
```

Se non conoscete il problema dello slicing, chiedete a Internet e al vostro motore di ricerca preferito: troverete molte informazioni. Scoprirete che l’esistenza del problema dello slicing è un altro motivo (oltre al problema dell’efficienza) per cui il passaggio per valore ha una pessima reputazione in C++98. Vi sono buoni motivi per cui una delle prime cose che si impara sulla programmazione in C++ era quella di evitare di passare per valore oggetti di tipi definiti dall’utente.

Il C++11 non cambia sostanzialmente il comportamento del C++98 relativamente al passaggio per valore. In generale, il passaggio per valore

prevede comunque un degrado prestazionale che sarebbe il caso di evitare e il passaggio per valore può comunque condurre al problema dello slicing. La novità del C++11 è la distinzione fra gli argomenti lvalue ed rvalue. Le funzioni di implementazione che sfruttano le semantiche di spostamento per gli rvalue di tipi copiabili richiedono o l'overloading o l'utilizzo di riferimenti universali, entrambe soluzioni che presentano difetti. Per il caso speciale dei tipi copiabili, economici da spostare, passati a funzioni che li copiano sempre e dove il problema dello slicing non è percepibile, il passaggio per valore può offrire un'alternativa di facile implementazione che è quasi altrettanto efficiente del passaggio per riferimento, ma attenzione agli svantaggi.

Argomenti da ricordare

- Per i parametri copiabili ed economici da spostare che vengono sempre copiati, il passaggio per valore può essere efficiente quasi quanto il passaggio per riferimento, ma è più facile da implementare e può generare meno codice oggetto.
- La copia di parametri per costruzione può essere molto più costosa rispetto alla copia per assegnamento.
- Il passaggio per valore è soggetto al problema dello slicing e, pertanto, in generale è inappropriato per i tipi di parametri della classe base.

Elemento 42 – Impiegare l'emplacement al posto dell'inserimento

Se avete un container che contiene, per esempio, delle `std::string`, sembra logico che quando aggiungete un nuovo elemento tramite una funzione di inserimento (per esempio `insert`, `push_front`, `push_back` o, per `std::forward_list`, `insert_after`), il tipo di elemento da passare alla funzione sarà `std::string`. Dopotutto, è proprio questo il contenuto del container.

Per quanto possa sembrare logico, non è sempre così. Considerate il codice seguente:

```
std::vector<std::strina> vs;      // container di std::string

vs.push_back(“xyzzy”);            // aggiunta di una stringa
                                  letterale
```

Qui, il container contiene certamente delle `std::string`, ma ciò che si ha a disposizione, quando si esegue effettivamente il `push_back`, è una stringa letterale, ovvero una sequenza di caratteri racchiusi fra doppi apici. Una stringa letterale non è una `std::string` e ciò significa che l'argomento passato a `push_back` non è il tipo contenuto nel container.

`push_back` per `std::vector` subisce overloading per lvalue ed rvalue nel seguente modo:

```
template <class T,                // dallo standard del C++11
      class Allocator =
      allocator<T>>
class vector {
public:
  …
  void push_back(const T& x);     // inserisce un lvalue
  void push_back(T&& x);          // inserisce un rvalue
  …
};
```

Nella chiamata:

```
vs.push_back("xyzzy");
```

i compilatori trovano una differenza tra il tipo dell'argomento (`const char[6]`) e il tipo del parametro preso da `push_back` (un riferimento a una `std::string`). Il modo in cui risolvono questa differenza di tipo consiste nella generazione di codice per creare un oggetto temporaneo `std::string` utilizzando la stringa letterale, per poi passare tale oggetto temporaneo a `push_back`. In altre parole, trattano la chiamata come se fosse scritta nel seguente modo:

```
vs.push_back(std::string("xyzzy")); // crea una std::string
                                    temporanea
                                    // e la passa a push_back
```

Il codice viene compilato e funziona e quindi tutti sono “felici”. Tutti tranne i maniaci delle prestazioni, che si accorgono che questo codice non è efficiente quanto potrebbe.

Per creare un nuovo elemento in un container di `std::string`, pensano, dovrebbe essere richiamato un costruttore di `std::string`, ma il codice precedente non esegue una semplice chiamata al costruttore. Ne esegue due. E richiama anche il distruttore di `std::string`. Ecco che cosa accade runtime con la chiamata a `push_back`.

1. Viene creato un oggetto temporaneo di tipo `std::string` a partire dalla stringa letterale “xyzzy”. Questo oggetto non ha alcun nome, ma lo chiameremo *`temp`*. La costruzione di *`temp`* è la prima costruzione di una `std::string`. Poiché si tratta di un oggetto temporaneo, *`temp`* è un rvalue.

2. *`temp`* viene passato all’overloading per rvalue di `push_back`, dove viene associato al parametro riferimento rvalue `x`. Una copia di `x` viene poi costruita in memoria per lo `std::vector`. Tale costruzione, la *seconda*, è quella che crea effettivamente un nuovo oggetto all’interno dello `std::vector`. (Il costruttore utilizzato per copiare `x` nello `std::vector` è il costruttore per spostamento, poiché `x`, essendo un riferimento rvalue, viene convertito in un rvalue prima di essere copiato. Per informazioni sulla conversione in rvalue dei parametri riferimento rvalue, vedere l’Elemento 25.)

3. Immediatamente dopo l’uscita da `push_back`,*`temp`* viene distrutta, richiedendo una chiamata al distruttore di `std::string`.

I maniaci delle prestazioni non possono fare a meno di notare che se vi fosse un modo per prendere la stringa letterale e passarla direttamente al codice nel passo 2 che costruisce l’oggetto `std::string` all’interno dello `std::vector`, si potrebbe evitare del tutto di costruire e distruggere *`temp`*. Questo meccanismo sarebbe particolarmente efficiente e placherebbe l’ansia del maniaco delle prestazioni.

Poiché siete programmatori C++, è piuttosto probabile che siate anche voi maniaci delle prestazioni. Se non lo siete, proverete comunque una certa sintonia con questo punto di vista (e se non siete affatto interessati alle prestazioni, potete sempre accomodarvi fra gli appassionati di Python). Ho il piacere di dirvi che un

modo per fare esattamente quanto è richiesto e ottenere la massima efficienza nella chiamata a `push_back` esiste. La soluzione consiste nel non chiamare `push_back`. `push_back` è la funzione sbagliata. La funzione da utilizzare è `emplace_back`.

`emplace_back` fa esattamente ciò di cui avete bisogno: utilizza gli argomenti che le passate per costruire una `std::string` direttamente all'interno dello `std::vector`. Ma senza richiedere la creazione di oggetti temporali:

```
vs.emplace_back("xyzzy");          // costruisce la std::string in
                                   // vs usando direttamente "xyzzy"
```

`emplace_back` usa il perfect-forward e pertanto, sempre che non vi scontriate con i suoi limiti (vedi Elemento 30), permette di passare un qualsiasi numero di argomenti di qualsiasi combinazione di tipi. Per esempio, se voleste creare una `std::string` in `vs` tramite il costruttore di `std::string` che accetta un carattere e un contatore delle ripetizioni, potete farlo con:

```
vs.emplace_back(50, 'x');          // inserisce una std::string
                                   costituita da
                                   // 50 caratteri 'x'
```

`emplace_back` è disponibile per ogni container standard che supporti `push_back`. Analogamente, ogni container standard che supporta `push_front` supporta anche `emplace_front`. E ogni container standard che supporta `insert` (ovvero tutti tranne `std::forward_list` e `std::array`) supporta `emplace`. I container associativi offrono `emplace_hint` per affiancare le proprie funzioni `insert` che accettano un iteratore di "suggerimento" e `std::forward_list` ha un `emplace_after` da utilizzare al posto di `insert_after`.

Ciò che consente alle funzioni di emplacement di battere in prestazioni le funzioni di inserimento è la loro interfaccia più flessibile. Le funzioni di inserimento prendono gli *oggetti da inserire*, mentre le funzioni di emplacement prendono gli *argomenti di tipo costruttore per gli oggetti da inserire*. Questa differenza permette alle funzioni di emplacement di evitare di creare e distruggere degli oggetti temporanei, così come richiesto dalle funzioni di inserimento.

Poiché a una funzione di emplacement può essere passato un argomento del tipo contenuto nel container (l'argomento fa pertanto in modo che la funzione svolga

la costruzione per copia o per spostamento), l’emplacement può essere utilizzato anche quando la funzione di inserimento non richiederebbe alcun oggetto temporaneo. In tal caso, le funzioni di inserimento e di emplacement si trovano a svolgere praticamente la stessa operazione. Per esempio, data

```
std::string queenOfDisco("Donna Summer");
```

entrambe le seguenti chiamate sono valide ed entrambe hanno lo stesso effetto sul container:

```
vs.push_back(queenOfDisco);       // costruzione per copia di queenOfDisco
                                  // alla fine di vs
vs.emplace_back(queenOfDisco);    // la stessa cosa
```

Le funzioni di emplacement, pertanto, possono fare tutto ciò che fanno le funzioni di inserimento. Talvolta si comportano in modo più efficiente e, almeno in teoria, non dovrebbero mai essere meno efficienti. Allora, perché non utilizzarle sempre?

Perché, come si suol dire, in teoria non vi è alcuna differenza fra la teoria e la pratica, ma in pratica, tale differenza esiste. Con le implementazioni attuali della Libreria Standard, vi sono situazioni in cui, come previsto, le operazioni di emplacement si comportano molto meglio delle funzioni di inserimento, ma, purtroppo, vi sono situazioni in cui le funzioni di inserimento risultano più veloci. Tali situazioni non sono facili da individuare, poiché dipendono dai tipi di argomenti passati, dai container utilizzati, dalla posizione in cui si trova il container quando viene richiesta l’operazione di inserimento o di emplacement, dalla sicurezza alle eccezioni dei costruttori dei tipi contenuti e, per i container in cui sono vietati valori duplicati (per esempio `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`), dal fatto che il valore da aggiungere sia già presente nel container. Si applica pertanto il solito consiglio relativo alle prestazioni: per determinare se si comporta più velocemente l’emplacement o l’inserimento, occorre valutarne le prestazioni.

La risposta non è molto soddisfacente, naturalmente, quindi sarete felici di sapere che esiste un’euristica che può aiutarvi a identificare le situazioni in cui le funzioni di emplacement sono particolarmente efficaci. Se valgono i seguenti fatti, le operazioni di emplacement saranno quasi certamente molto migliori rispetto alle funzioni di inserimento.

- **Il valore da aggiungere viene costruito nel container, non assegnato.** L'esempio che ha aperto questo Elemento (l'aggiunta di una `std::string` con il valore "`xyzzy`" a un `std::vector` di nome `vs`) ha mostrato come il valore venga aggiunto alla fine di `vs`, in una posizione in cui non esisteva ancora alcun oggetto. Il nuovo valore, pertanto, doveva essere costruito all'interno di `std::vector`. Se elaboriamo l'esempio in modo che la nuova `std::string` entri in una posizione già occupata da un oggetto, è tutta un'altra faccenda. Considerate il seguente esempio:

```
std::vector<std::string> vs;      // come prima

…                                 // aggiunge elementi a vs

vs.emplace(vs.begin(), "xyzzy");  // aggiunge "xyzzy"
                                  // all'inizio di vs
```

Per questo codice, poche implementazioni costruiranno la `std::string` aggiunta nella memoria occupata da `vs[0]`. Piuttosto, assegneranno per spostamento il valore in loco. Ma l'assegnamento per spostamento richiede un oggetto dal quale eseguire lo spostamento e ciò significa che dovrà essere creato un oggetto temporaneo che funga da origine dello spostamento. Poiché il vantaggio principale dell'emplacement rispetto all'inserimento è il fatto che non vengono né creati né distrutti oggetti temporanei, quando il valore da aggiungere viene inserito nel container tramite un assegnamento, il punto di forza dell'emplacement tende a sparire.

Purtroppo, il fatto che l'aggiunta di un valore a un container venga conseguita per costruzione o per assegnamento dipende generalmente dall'implementatore. Ma, ancora una volta, può essere d'aiuto l'euristica. I container basati su nodi utilizzano praticamente sempre la costruzione per aggiungere nuovi valori e la maggior parte dei container standard è basata su nodi. Gli unici a non esserlo sono `std::vector`, `std::deque` e `std::string` (neppure `std::array` lo è, ma non supporta né l'inserimento né l'emplacement e quindi non è rilevante per questa discussione). Parlando di container non basati su nodi, potete contare su `emplace_back` per utilizzare la costruzione al posto dell'assegnamento per collocare un nuovo valore in loco e, per `std::deque`, vale la stessa cosa per `emplace_front`.

- **I tipi di argomenti passati differiscono dal tipo contenuto nel container.** Ancora una volta, il vantaggio dell'emplacement rispetto all'inserimento deriva generalmente dal fatto che la sua interfaccia non richiede la creazione e la distruzione di un oggetto temporaneo quando gli argomenti passati sono di un tipo diverso da quello contenuto nel container. Quando a un `container<T>` deve essere aggiunto un oggetto di tipo `T`, non vi è alcun motivo per aspettarsi che l'emplacement risulti più veloce rispetto all'inserimento, poiché non deve essere creato alcun oggetto temporaneo per soddisfare l'interfaccia di inserimento.

- **È improbabile che il container rifiuti il nuovo valore come un duplicato.** Questo significa che il container o permette l'esistenza di duplicati oppure si sa che la maggior parte dei valori inseriti sarà univoca. Questo fattore è importante, perché per rilevare se un valore è già presente nel container, le implementazioni di emplacement creano tipicamente un nodo con il nuovo valore, in modo che si possa confrontare il valore di questo nodo con i nodi esistenti nel container. Se il valore da aggiungere non è nel container, il nodo viene collegato. Al contrario, se il valore è già presente, l'emplacement viene annullato e il nodo viene distrutto, facendo sì che il costo della sua costruzione e distruzione sia completamente sprecato. Tali nodi vengono creati per le funzioni di emplacement più frequentemente rispetto alle funzioni di inserimento.

Le seguenti chiamate, utilizzate in precedenza in questo Elemento, soddisfano tutti i criteri precedenti. Operano anche più velocemente rispetto alle corrispondenti chiamate a `push_back`.

```
vs.emplace_back("xyzzy");          // costruisce un nuovo valore
                                   alla fine del
                                   // container; non passa il tipo
                                   nel
                                   // container; non usare container
                                   // che rifiutano duplicati

vs.emplace_back(50, 'x');          // la stessa cosa
```

Quando si deve decidere se utilizzare le funzioni di emplacement, vale la pena di considerare altri due fattori. Il primo riguarda la gestione delle risorse.

Supponete di avere un container di `std::shared_ptr<Widget>`

```
std::list<std::shared_ptr<Widget>> ptrs;
```

e di volergli aggiungere uno `std::shared_ptr` che dovrebbe essere rilasciato tramite un cancellatore custom (vedi Elemento 19). L'Elemento 21 spiega che si dovrebbe utilizzare `std::make_shared` per creare degli `std::shared_ptr` ogni volta che sia possibile, ma ammette anche che vi sono situazioni in cui ciò non è possibile. Una di queste situazioni si verifica quando si vuole specificare un cancellatore custom. In tale caso, occorre utilizzare `new` direttamente, per ottenere il puntatore standard che verrà gestito dallo `std::shared_ptr`.

Se il cancellatore custom è la seguente funzione,

```
void killWidget(Widget* pWidget);
```

il codice che utilizza una funzione di inserimento potrebbe avere il seguente aspetto:

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

Potrebbe inoltre avere il seguente aspetto, e il significato sarebbe lo stesso:

```
ptrs.push_back({ new Widget, killWidget });
```

Comunque, prima di richiamare `push_back` verrebbe costruito uno `std::shared_ptr` temporaneo. Il parametro di `push_back` è un riferimento a uno `std::shared_ptr`, pertanto vi deve essere uno `std::shared_ptr` cui può fare riferimento questo parametro.

La creazione dello `std::shared_ptr` temporaneo è ciò che `emplace_back` tenderebbe a evitare, ma in questo caso vale proprio la pena di creare tale oggetto temporaneo. Considerate la seguente catena di eventi potenziali.

1. In una delle chiamate precedenti, viene costruito un oggetto `std::shared_ptr<Widget>` temporaneo, per contenere il puntatore standard prodotto da `new Widget`. Chiamiamo quest'oggetto *`temp`*.

2. `push_back` prende *`temp`* per riferimento. Durante l'allocazione di un nodo della lista per contenere una copia di *`temp`*, viene lanciata un'eccezione di esaurimento della memoria.

3. Mentre l'eccezione si propaga fuori da `push_back`, *`temp`* viene distrutto.

Essendo l'unico `std::shared_ptr` che fa riferimento al `Widget` che sta gestendo, rilascia automaticamente tale `Widget`, in questo caso richiamando `killWidget`.

Anche se si è verificata un'eccezione, non si spreca nulla: il `Widget` creato tramite "`newWidget`" nella chiamata `push_back` viene rilasciato nel distruttore dello `std::shared_ptr` che era stato creato per gestirlo (ovvero *`temp`*). Tutto bene.

Ora considerate ciò che accade se venisse chiamata `emplace_back` invece di `push_back`:

```
ptrs.emplace_back(new Widget, killWidget);
```

1. Il puntatore standard prodotto da `newWidget` viene inviato in perfect-forward al punto all'interno di `emplace_back` dove deve essere allocato nel nodo della lista. Tale allocazione non ha successo e viene lanciata un'eccezione di esaurimento della memoria.

2. Mentre l'eccezione si propaga fuori da `emplace_back`, il puntatore standard che rappresentava l'unico modo per raggiungere il `Widget` sullo heap non esiste più. Tale `Widget` (e tutte le risorse che possiede) risulterà irraggiungibile.

In questa situazione, le cose non vanno per niente bene e la cosa non riguarda `std::shared_ptr`. Lo stesso genere di problema può sorgere tramite l'uso di `std::unique_ptr` con un cancellatore custom. Fondamentalmente, l'efficacia delle classi di gestione delle risorse come `std::shared_ptr` e `std::unique_ptr` conta sul fatto che tali risorse (come i puntatori standard prodotti da `new`) vengano passate *immediatamente* ai costruttori per gli oggetti di gestione delle risorse. Il fatto che funzioni come `std::make_shared` e `std::make_unique` automatizzano tutto ciò è uno dei motivi per cui sono così importanti.

Nelle chiamate alle funzioni di inserimento dei container che contengono oggetti per la gestione delle risorse (come `std::list<std::shared_ptr<Widget>>`), i tipi dei parametri delle funzioni generalmente garantiscono che nulla si frapponga fra l'acquisizione di una risorsa (ovvero l'uso del `new`) e la costruzione dell'oggetto che gestisce tale risorsa. Nelle funzioni di emplacement, il perfect-forward rimanda la creazione degli oggetti di gestione delle risorse fino al momento in cui possono essere costruiti nella memoria del container e ciò apre una finestra temporale entro la quale le eccezioni possono causare uno

spreco di risorse. Tutti i container standard sono sensibili a questo problema. Quando si lavora con i container di oggetti per la gestione delle risorse, occorre fare attenzione per garantire che, se si sceglie una funzione di emplacement al posto della sua controparte per inserimento, non si paga una migliore efficienza del codice con una minore sicurezza di gestione delle eccezioni.

Francamente, non si dovrebbero passare a `emplace_back` o `push_back` o alla maggior parte delle altre funzioni, espressioni `new Widget`, poiché, come spiega l'Elemento 21, questo porta a potenziali problemi di sicurezza di gestione delle eccezioni, del tipo già esaminato. Per chiudere la porta a questo problema, occorre prendere il puntatore da `new Widget` e trasformarlo in un oggetto di gestione delle risorse all'interno di un'istruzione indipendente, per poi passare tale oggetto come un rvalue alla funzione alla quale si voleva passare originariamente `new Widget` (l'Elemento 21 descrive più in dettaglio questa tecnica). Il codice che utilizza `push_back` dovrebbe pertanto essere scritto nel seguente modo:

```
std::shared_ptr<Widqet> spw(new       // crea il Widget e
Widget,
        killWidget);                  // lo fa gerstire da spw

ptrs.push_back(std::move(spw));       // aggiunge spw come rvalue
```

La versione `emplace_back` è simile:

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

In ogni caso, l'approccio incorre nel costo della creazione e distruzione di `spw`. Dato che la motivazione per scegliere l'emplacement rispetto all'inserimento consiste nell'evitare il costo di un oggetto temporaneo del tipo contenuto dal container, ovvero ciò che è teoricamente `spw`, le funzioni di emplacement difficilmente batteranno in prestazioni le funzioni di inserimento quando si devono aggiungere oggetti di gestione delle risorse a un container e si adotta la pratica corretta: garantire che nulla possa frapporsi fra l'acquisizione di una risorsa e la sua trasformazione in un oggetto di gestione delle risorse.

Un secondo aspetto degno di nota delle funzioni di emplacement è rappresentato dalla loro interazione con i costruttori `explicit.` In onore al supporto C++11 per le espressioni regolari, supponete di creare un container di oggetti che sono

espressioni regolari:

```
std::vector<std::regex> regexes;
```

Distratti dai vostri colleghi che discutono sul numero ideale di volte in cui si dovrebbe fare accesso a Facebook quotidianamente, scrivete accidentalmente il seguente codice, apparentemente senza senso:

```
regexes.emplace_back(nullptr);    // aggiunge nullptr al container
                                  // di regex?
```

Non notate l'errore mentre lo digitate e il compilatore accetta il codice senza battere ciglio, così vi trovate a sprecare ore e ore di debugging. A un certo punto, scoprite di aver inserito un puntatore nullo nel vostro container di espressioni regolari. Ma come è possibile? I puntatori non sono espressioni regolari e se tentate di fare qualcosa come,

```
std::regex r = nullptr;           // errore! incompilabile
```

i compilatori rifiuteranno il codice. È interessante notare che lo rifiuterebbero anche se aveste chiamato `push_back` invece che `emplace_back`:

```
regexes.push_back(nullptr);       // errore! incompilabile
```

Il comportamento curioso che state sperimentando deriva dal fatto che gli oggetti `std::regex` possono essere costruiti partendo da stringhe di caratteri. È proprio ciò che rende legale del codice utile come il seguente:

```
std::regex upperCaseWord("[A-Z]+");
```

La creazione di uno `std::regex` da una stringa di caratteri può produrre un costo runtime piuttosto esteso e pertanto, per minimizzare la probabilità che si verifichi inavvertitamente tale spesa, il costruttore di `std::regex` che accetta un puntatore `const char*` è `explicit`. Questo è il motivo per cui le seguenti righe non possono essere compilate:

```
std::regex r = nullptr;           // errore! incompilabile

regexes.push_back(nullptr);       // errore! incompilabile
```

In entrambi i casi, stiamo chiedendo una conversione implicita da un puntatore a

uno `std::regex` e il fatto che il costruttore sia esplicito impedisce tali conversioni.

Nella chiamata a `emplace_back`, invece, non pretendiamo di passare un oggetto `std::regex`. Al contrario, passiamo un *argomento costruttore* per un oggetto `std::regex`. Questa non viene considerata una richiesta di conversione implicita. Piuttosto, è come se aveste scritto il seguente codice:

```
std::regex r(nullptr);            // compilabile
```

Se il commento laconico, "`compilabile`", suggerisce una mancanza di entusiasmo, è proprio così, poiché questo codice, anche se è compilabile, ha un comportamento indefinito. Il costruttore di `std::regex` che accetta un puntatore a `const char*` richiede che la stringa puntata contenga un'espressione regolare valida e il puntatore nullo non risponde a questo requisito. Se scrivete e compilate questo codice, la cosa migliore che potrete sperare è che si blocchi runtime. Se non sarete così fortunati, voi e il vostro debugger potreste andare incontro a un'esperienza comune piuttosto lunga e per niente gradevole.

Mettendo da parte `push_back`, `emplace_back` e i problemi affettivi con il debugger, notate come queste sintassi di inizializzazione molto simili forniscano risultati differenti:

```
std::regex r1 = nullptr;          // errore! incompilabile

std::regex r2(nullptr);           // compilabile
```

Nella terminologia ufficiale dello Standard, la sintassi utilizzata per inizializzare r1 (impiegando il segno di uguaglianza) corrisponde a ciò che è chiamato *inizializzazione per copia*. Al contrario, la sintassi utilizzata per inizializzare r2 (con le parentesi, anche se si potrebbero utilizzare le graffe) fornisce ciò che viene chiamata *inizializzazione diretta*. L'inizializzazione per copia non può utilizzare costruttori `explicit`. L'inizializzazione diretta invece sì. Questo è il motivo per cui la riga che inizializza r1 non passa la compilazione, mentre la riga che inizializza r2 sì.

Ma torniamo a `push_back` ed `emplace_back` e, più in generale, alle funzioni di inserimento: rispetto alle funzioni di emplacement, queste ultime utilizzano l'inizializzazione diretta, che significa che possono utilizzare costruttori

`explicit`. Le funzioni di inserimento impiegano l'inizializzazione per copia e pertanto non possono farlo. Quindi:

```
regexes.emplace_back(nullptr);   // compilabile.
                                 L'inizializzazione diretta
                                 // permette l'uso di costr.
                                 std::regex
                                 // espliciti che accettano un
                                 puntatore

regexes.push_back(nullptr);      // errore! proibito
                                 // l'uso di questo costr.
```

La "morale"? Quando si usa una funzione di emplacement, occorre fare particolare attenzione ad assicurarsi di passare gli argomenti corretti, poiché anche i costruttori `explicit` verranno considerati dai compilatori come se tentassero di trovare un modo per interpretare il vostro codice come valido.

Argomenti da ricordare

- In linea di principio, le funzioni di emplacement dovrebbero essere più efficienti rispetto alla loro versione per inserimento e non dovrebbero mai essere meno efficienti.
- In pratica, saranno molto probabilmente più veloci quando (1) il valore aggiunto viene costruito nel container, non assegnato; (2) i tipi di argomenti passati differiscono dal tipo contenuto nel container; (3) il container non rifiuterà il valore aggiunto a causa di un duplicato già presente al suo interno.
- Le funzioni di emplacement possono eseguire conversioni di tipo che sarebbero rifiutate dalle funzioni di inserimento.

---

[22]. In questo Elemento, con "copiare" un parametro, si intende generalmente il suo uso come origine di un'operazione di copia o spostamento. Abbiamo detto nell'Introduzione che il C++ non offre una terminologia precisa per distinguere una copia eseguita da un'operazione di copia da una copia eseguita da

un’operazione di spostamento.

[23]. Frasi come questa fanno pensare a quanto sia bello avere una terminologia che distingue le copie eseguite tramite operazioni di copia e le copie eseguite tramite operazioni di spostamento.

Indice analitico

A addDivisorFilter alias
dichiarazioni
API per la concorrenza argomenti funzione array
argomenti
decade
asincronicità
authAndAccess
auto
deduzione del tipo modificatore
auto&&

B
basic_ios
bitfield
blocco di controllo Boost TypeIndex **C**
cachedValue
campi bit
cancellatori personalizzati cattura di default cattura esplicita classe closure
classi RAII
closure
coerenza sequenziale collasso dei riferimenti computePriority const
funzioni membro const char
constexpr
uso quando possibile const_iterator
const T&, passaggio container
conteggio dei riferimenti continueProcessing copia

conteggio dei riferimenti continueProcessing copia
costruzione per copia creazione di oggetti Curiously Recurring Template Pattern

D
dead stores
decltype
funzionamento
decltype(auto)
deduzione del tipo delete
detach implicito diagnostiche di compilazione dichiarazioni alias dichiarazioni esplicite del tipo distinzione fra parentesi tonde e graffe
doWork
durata di memorizzazione statica **E**
editor IDE
elisione della copia emplace_back
emplacement
enum
con visibilità
tipo intero base unscoped e scoped enum class
espressioni lambda eventi “one-shot”
expr

F
fastLoadWidget
funzione membro template funzione stateless funzioni cancellate funzioni che non emettono eccezioni funzioni con contratti ampi o stretti funzioni di emplacement funzioni membro speciali funzioni overloaded funzioni private undefined future void

G
Gadget
garbage-collection generazione di funzioni membro speciali
gestione delle risorse a proprietà condivisa

H
handle del thread **I** IDE
idioma Pimpl
inferenza del tipo init capture
inizializzatore a graffe inizializzatore di tipo esplicito inizializzazione a graffe

inizializzazione uniforme I/O mappate in memoria isLucky

J
join implicito

L
lambda generalizzata Libreria Standard lockAndCall
logAndAddImpl

M
make
makeInvestment
makeWidget
malfunzionamento Matrix
most vexing parse mutable
MyAllocList

N
new
noexcept
dichiarare le funzioni noexcept condizionali nomi template
nullptr
preferirlo a 0 e NULL

O
oggetti bind
oggetti RAII
operator delete operator new
operazione di spostamento ottimizzazione SSO
output runtime
overloading
abbandono
alternative per riferimenti universali
evitarlo sui riferimenti universali
override
dichiarare le funzioni overriding
condizioni

P
ParamType
passaggio per valore riferimento o puntatore riferimento universale parole chiave contestuali passaggio per const T& passaggio per valore perfect-forward perfect-forwarding Person
politica di lancio PolyWidget
pow
printf
private
problema dello slicing processPointer
processVal
programmazione basata su task proprietà condivisa proprietà esclusiva puntatori nulli puntatori smart push_back

Q
qualificatori reference **R**
ReallyBigType
redundant loads reference collapsing Regola dei tre
Return Value Optimization riferimenti universali riferimento universale risvegli spuri

S
scope
semantica di spostamento setAlarm
setName
setSoundL
sizeof(Widget)
spostamento non veloce static_assert
static_cast
static const
stato condiviso std::array
std::async
std::atomic
std::auto_ptr
std::bind
std::count_if
std::decay

std::deque std::enable_if
std::false_type std::find_if
std::forward
std::function
std::future
std::future_status::deferred std::get
std::initializer_list std::initializer_list.
std::is_constructible std::is_same
std::launch::async std::make_shared std::make_unique std::move
std::move(data) std::remove_if
std::remove_reference_t std::shared_ptr std::size_t
std::string
std::thread
std::thread non-joinable std::true_type
std::type_info
std::unique_ptr std::vector
std::vector<bool> std::vector<double> std::weak_ptr

T
tag dispatch
  approccio
task di reazione task di rilevamento tecniche varie
template
template alias
  scrittura del
thread hardware thread interrompibili thread software timeFuncInvocation tipi
  dedotti, visualizzazione tipo move-only
TLS (Thread-Local Storage) trasformazione algoritmica typedef
typeid
type trait

V
void
volatile

W
Widget
Widget&

Widget::addFilter Widget::Impl
Widget::magicValue Widget::MinVals Widget::process work-stealing
wrapper

Informazioni sul Libro

Per programmare in modo davvero efficace con i linguaggi C++11 e C++14 non basta sfogliare qualche nota introduttiva sulle nuove funzionalità (per esempio le dichiarazioni di tipo auto, le semantiche di spostamento, le espressioni lambda e il supporto della concorrenza). Occorre imparare a utilizzare queste nuove funzionalità in modo efficace, per produrre software che risulti corretto, efficiente, di facile manutenzione e portabile. Proprio a tale esigenza risponde questo manuale, che spiega in modo pratico quali tecniche impiegare per scrivere software davvero di alto livello utilizzando i linguaggi C++11 e C++14, ovvero il *C++ moderno.*

- I pro e i contro dell'inizializzazione a graffe, delle specifiche noexcept, del perfect forward e delle funzioni make per i puntatori smart ■Le relazioni esistenti fra std::move, std::forward, i riferimenti rvalue e i riferimenti universali ■Le tecniche per scrivere espressioni lambda chiare, corrette ed efficaci
- Le differenze fra std :: atomic e volatile, come devono essere utilizzati e come si relazionano con l'API per concorrenza del C++
- Fino a che punto le vecchie tecniche di programmazione C++ (ovvero per il C++98) devono essere riviste per sviluppare software in C++ moderno

Programmazione C++ moderna adotta il rinomato stile basato su indicazioni ed esempi, tipico dei libri di Scott Meyers, introducendo però materiale interamente nuovo. Una lettura fondamentale per ogni sviluppatore C++.

Circa l’autore

Per più di 20 anni, i libri Effective C++ di **Scott Meyers** (*Effective C++*, *More Effective C++* ed *Effective STL*) sono stati i testi di riferimento per la programmazione C++. Le sue spiegazioni chiare e avvincenti di materiale tecnico complesso sono apprezzate in tutto il mondo, tanto che Scott è sempre più richiesto come trainer, consulente e relatore.